Fachhochschule Köln
Cologne University of Applied Sciences

10 Fakultät für Informatik und
Ingenieurwissenschaften

Diplomarbeit

Entwurf und Implementierung einer effizienten Dublettenerkennung für große Adressbestände

vorgelegt an der

Fachhochschule Köln, Campus Gummersbach
im Studiengang Allgemeine Informatik

ausgearbeitet von

Thomas Krause

Erster Prüfer: Prof. Dr. Erich Ehses
Zweiter Prüfer: Prof. Dr. Heide Faeskorn-Woyke

Gummersbach, im Februar 2012

Zusammenfassung

Dublettenerkennung bezeichnet einen Prozess zur Ermittlung ähnlicher oder identischer Datensätze, die sich auf das gleiche Objekt der realen Welt beziehen. Eine besondere Bedeutung hat diese im Rahmen einer Datenbereinigung zum Beispiel nach dem Zusammenführen verschiedener Datenbestände mit möglichen Überlappungen.

In diesem Zusammenhang haben sich in den letzten Jahren einige interessante Entwicklungen ergeben. Zum einen steigen die erfassten Datenmengen immer weiter an, so dass Algorithmen an Bedeutung gewinnen, die auch in solchen großen Datenbeständen effizient arbeiten. Zum anderen steigt durch die stärkere Verbreitung von Mehrkernprozessoren und die zunehmende Zentralisierung von IT-Diensten (vgl. Cloud Computing) auch der Bedarf an Lösungen, die in solchen Umgebungen optimal arbeiten und sich gut skalieren lassen.

Der hier vorgestellte Lösungsansatz zur Dublettenerkennung kombiniert einen modernen und effizienten Algorithmus mit den Vorzügen einer zentralen und dienstorientierten Architektur.

Abstract

Duplicate detection is the process to identify similar or identical records, that represent the same real world entity. This has special importance in the data cleansing process; e.g. after merging different data sets which may overlap.

Related to this there have been some interesting developments in recent years. On one hand the typical data sets continue to grow which requires efficient detection algorithms, which can support these large data sets. On the other hand the distribution of multi-core processors and the continued centralization of IT services demand solutions that can work optimally in these environments and are scalable.

The solution to duplicate detection presented in this paper combines a modern and efficient algorithm with the advantages of a centralized and service-oriented architecture.

Inhaltsverzeichnis

Abbildungsverzeichnis

Notation

Diese Liste gibt eine Übersicht über in der Arbeit häufig verwendete Symbole und ihrer Bedeutung:

Ψ	Menge aller Datensätze (Datenbestand)
$\|\Psi\|$	Anzahl der Datensätze
$A^{\#}, B^{\#}, ...$	Einzelne Datensätze
T	Die Menge aller Token im Datenbestand
t	Ein Token aus T
A	Die Liste der Token im Datensatz $A^{\#}$ (= Dokument)
Λ	Die Menge aller Dokumente (Dokumentkorpus)
A^{+}	Der Dokumentvektor zum Datensatz $A^{\#}$
F	Die Menge aller Felder (Attribute)
σ	Ein Schwellwert für die Ähnlichkeit
ν	Ein Schwellwert für die Reliabilität
$tf(D^{\#}, t)$	Anzahl der Vorkommen des Token t im Datensatz $D^{\#}$

1 Einleitung

1.1 Bedeutung

Als Dubletten werden im IR (Information Retrieval) zwei Datensätze bezeichnet, die sich auf das gleiche Objekt der realen Welt (Entität) beziehen.

Dublettenerkennung bezeichnet den Prozess zur Ermittlung solcher Dubletten. Eine besondere Bedeutung hat diese im Rahmen einer Datenbereinigung; zum Beispiel nach dem Zusammenführen verschiedener Datenbestände mit möglichen Überlappungen.

1.2 Motivation

Dubletten in Datenbeständen verursachen einen großen Schaden in der Realwirtschaft. Erhält ein Kunde mehrmals das gleiche Anschreiben eines Unternehmens, weil er dort mehrfach im Adressbestand geführt wird, so kostet dies das Unternehmen nicht nur zusätzliches Geld für Verpackung und Versand, sondern es verärgert womöglich auch einen ansonsten zufriedenen Kunden. Statistische Kennzahlen wie Kundenanzahl oder mittlerer Umsatz pro Kunde werden durch Dubletten verfälscht, da einzelne Kunden mehrmals gezählt werden beziehungsweise der Umsatz auf mehrere, eigentlich identische, Kunden aufgeteilt wird (vgl. Leser und Naumann 2007, Seite 329ff).

Nur selten bestehen Dubletten auch aus identischen Datensätzen. Solche exakten Duplikate werden oft schon auf Datenbankebene erkannt und zurückgewiesen. Dubletten enthalten also oft fehlerhafte oder abweichende Darstellungen der gleichen Informationen. Die Herausforderung der Dublettenerkennung besteht darin, auch solche Dubletten zu erkennen, die eine unterschiedliche Darstellung besitzen.

Eng verwandt mit der Erkennung von Dubletten ist die Suche in Datenbeständen. Man kann die Dublettenerkennung als Spezialfall der allgemeinen Suche verstehen, bei der für einen bestimmten Datensatz

$D^{\#}$ übereinstimmende oder ähnliche Datensätze im Datenbestand gesucht werden. Dementsprechend gibt es auch bei den verwendeten Algorithmen große Überschneidungen und die dort verwendeten Optimierungen lassen sich zum Teil auf das Gebiet der Dublettenerkennung übertragen. Interessante Möglichkeiten ergeben sich, wenn Techniken aus der allgemeinen Suche mit klassischen Methoden der Dublettenerkennung kombiniert werden, um die jeweiligen Nachteile auszugleichen oder zumindest abzuschwächen.

Wenn ein effektiver (gute Ergebnisse) und effizienter (hohe Geschwindigkeit) Algorithmus entwickelt wurde, so stellt die Implementierung dieses Algorithmus in ein produktionsreifes System eine weitere Herausforderung dar. Ein gutes asymptotisches Laufzeitverhalten hilft zum Beispiel wenig, wenn die tatsächliche Laufzeit für die gewünschte Eingabegröße nicht zufriedenstellend ist. Es müssen also weitere Kriterien betrachtet werden, um einen Algorithmus zu bewerten. Gute Algorithmen sollten zum Beispiel die Fähigkeiten von Mehrkernprozessoren ausnutzen können, um so die Geschwindigkeit noch um ein Vielfaches zu steigern.

Da Dublettenerkennung letztendlich nur Mittel zum Zweck ist, stellt die einfache Integration in bestehende Systeme eine wichtige Eigenschaft dar. Binärbibliotheken sind hier problematisch, da sie (Entwicklungs-) Plattform-spezifisch sind und sich deshalb nicht direkt in jedem System einsetzen lassen. Der Einsatz eines eigenständigen Servers bzw. Dienstes zur Erkennung und Weiterverarbeitung von Dubletten löst dieses Problem, erfordert aber eine leicht zu nutzende, robuste Schnittstelle zwischen nutzender Anwendung und dem Dienst.

1.3 Zielsetzung und Abgrenzung

In den folgenden Kapiteln soll ein System vorgestellt werden, dass die Erkennung von Dubletten in mittleren bis großen Datenbeständen ermöglicht. Ziel ist dabei – neben der hohen Erkennungsrate – eine gute Skalierbarkeit der zugrundeliegenden Algorithmen bei gleichzeitig einfacher Konfiguration für den Benutzer. Um dies zu erreichen, sollen altbekannte Techniken aus der Dublettenerkennung mit den modernen Erkenntnissen aus dem IR und insbesondere dem Bereich der Suchma-

schinen verknüpft werden. Dabei wird gezeigt, wie die Effizienz eines solchen Systemes weiter gesteigert werden kann, ohne die Erkennungsleistung zu beeinträchtigen. Das Ergebnis ist ein vollständiges System, das als Web-Dienst Datensätze entgegennehmen, verarbeiten und anschließend eine Liste der mutmaßlichen Dubletten zurückgeben kann. Die Implementierung des Systems als Web-Dienst über standardisierte Protokolle ermöglicht die einfache Integration in bestehende Systeme.

Explizit nicht abgedeckt, wird die Verarbeitung der Ergebnisse, also das Zusammenführen oder Löschen von Dubletten im Zielsystem, da dies in der Regel viel domänenspezifisches Wissen erfordert und sich schwer verallgemeinern lässt.

1.4 Gliederung

Die restliche Arbeit gliedert sich wie folgt:

Im Kapitel „**Technische Grundlagen**“ werden technische Grundlagen erläutert, die für das Verständnis der folgenden Kapitel essenziell sind. Dazu gehört auch eine Übersicht über vorangegangene Arbeiten in diesem Bereich.

Das Kapitel „**Entwurf**“ beschreibt und begründet die verwendeten Algorithmen und Techniken, die in dem hier vorgestellten System angewendet wurden.

In dem Kapitel „**Implementierung**“ wird dann die konkrete Implementierung dieses Entwurfs in Ausschnitten beschrieben.

Die Leistung des Systems wird im Kapitel „**Leistungsbewertung**“ näher betrachtet.

Das letzte Kapitel „**Ausblick und Zusammenfassung**“ fasst die Ergebnisse dieser Arbeit noch einmal zusammen und gibt Anregungen für mögliche zukünftige Erweiterungen und Verbesserungen.

2 Technische Grundlagen

Dieses Kapitel soll einen Überblick über die theoretischen Grundlagen der Dublettenerkennung bis hin zu auf dem Markt verfügbaren Produkten bieten. Dazu werden zunächst einige grundlegende Begriffe definiert und anschließend der Stand der Technik in der Dublettenerkennung dargestellt.

2.1 Entstehung von Dubletten

Ab einer gewissen Größe lässt sich ein Datenbestand nicht mehr ohne Weiteres überblicken, so dass zum Beispiel ein Mitarbeiter, der neue Daten einpflegt, nicht im Vorhinein weiß, ob die Daten eventuell schon existieren. Der Mitarbeiter könnte in diesem Fall möglicherweise über eine integrierte Suchfunktion nach ähnlichen Datensätzen suchen. Dies ist jedoch sehr zeitaufwändig und womöglich verzichtet er im Zweifelsfall darauf, wenn er davon ausgeht, dass es sich bei den neuen Daten nicht um Dubletten handelt.

Ein anderes typisches Beispiel ist das Zusammenführen mehrerer Datenbestände. Auch hier kann es leicht zu Dubletten kommen, wenn es Überschneidungen zwischen den Datenbeständen gibt. In vielen Firmen ist dies ein wiederkehrender Vorgang, da z.B. Adressdaten von externen Anbietern dazugekauft werden.

Selbst wenn bereits eine rudimentäre Dublettenerkennung existiert, die das doppelte Einfügen von Datensätzen verhindern soll, kann es vorkommen, dass die Datensätze – obwohl sie sich auf das gleiche Objekt beziehen – Abweichungen oder sogar Fehler enthalten und daher bei einem simplen Dublettencheck nicht erkannt werden. Einige solcher Abweichungen beziehungsweise Fehler, die sich in die Datensätze einschleichen können und die Dublettenerkennung erschweren, sollen hier zusammengefasst werden:

Inhaltliche oder semantische Fehler bezeichnen im System gespeicherte Daten, die nicht (mehr) der Realität entsprechen. Ursachen

können eine falsche Erfassung oder auch veraltete Daten sein (z.B. nach einem Umzug).

Syntaktische Fehler stellen Verletzungen des logischen Schemas dar. Dies sind beispielsweise falsch zugeordnete Felder (Name mit Vorname vertauscht), also an sich korrekte Daten, die aber falsch strukturiert sind.

Lexikalische und phonetische Fehler entstehen aus Unwissenheit oder Missverständnis über die korrekte Schreibweise eines Datums. Besondere Gefahrenquellen sind gleichlautende Namen mit unterschiedlicher Schreibweise („Schmidt“ oder „Schmitt“).

Typographische Fehler sind unbeabsichtigte, meist mechanisch bedingte Fehler. Derartige Fehler äußern sich in ausgelassenen, zusätzlichen oder vertauschten Zeichen. Ein Beispiel dafür ist das unbeabsichtigte Drücken von mehreren Tasten gleichzeitig auf einer Tastatur.

Kodierungsfehler entstehen bei der Speicherung der Daten auf ein physisches Medium. Hierbei müssen die Daten kodiert, also die einzelnen Felder und Daten in Bits und Bytes konvertiert werden. Bei uneinheitlicher Kodierung kann es vorkommen, dass Daten, die in einer bestimmten Kodierung geschrieben wurden, in einer anderen Kodierung geladen werden. Besonders problematisch ist hier die Text- bzw. Zeichenkodierung, da es hier viele verschiedene Varianten gibt, die sich zudem zum Teil überschneiden, so dass Fehler oft erst spät oder gar nicht entdeckt werden (Beispiel: westeuropäische Kodierung vs. UTF-8).

Um auch solche Dubletten zu erkennen, müssen fehlertolerante Verfahren entwickelt werden, die mit derartigen Abweichungen umgehen können.

2.2 Definition einer Dublette und Ähnlichkeit

Bevor die technischen Aspekte der Dublettenerkennung genauer betrachtet werden, muss zunächst geklärt werden, was genau eine Dublette ausmacht. Formal gesehen sind zwei unterschiedliche Datensätze $A^{\#}$

und $B^{\#}$ aus einer Menge Ψ genau dann (und nur dann) Dubletten, wenn sie die gleiche Entität in der realen Welt repräsentieren. Mathematisch ausgedrückt:

$$IsDuplicate(A^{\#}, B^{\#}) \Leftrightarrow Entity(A^{\#}) = Entity(B^{\#}) \tag{2.1}$$

$$\text{für } A^{\#}, B^{\#} \in \Psi \wedge A^{\#} \neq B^{\#} \tag{2.2}$$

In der Realität ist es aufgrund unzureichender oder fehlender Daten nicht immer eindeutig erkennbar, ob zwei Datensätze die gleiche Entität darstellen. Es kann also nur mit einer gewissen Wahrscheinlichkeit gesagt werden, ob es sich bei zwei Datensätzen um Dubletten handelt oder nicht. Wichtigstes Indiz ist dabei, wie stark sich die betroffenen Datensätze ähneln. Da Ähnlichkeit kein fest definierter Begriff ist, muss zunächst ein Ähnlichkeitsmaß definiert werden, das zwei Datensätzen einen festen Ähnlichkeitswert zuordnet. Der Einfachheit halber wird definiert, dass dieser Wert im Bereich [0..1] liegt, wobei höhere Werte eine höhere Ähnlichkeit indizieren; der maximale Wert 1 besagt also, dass zwei Datensätze (unter diesem Ähnlichkeitsmaß) als identisch angesehen werden[1]. Für ein solches Ähnlichkeitsmaß sollen die folgenden Regeln gelten:

$$sim(A^{\#}, B^{\#}) \in [0..1] \qquad \text{(Beschränktheit)} \tag{2.3}$$

$$sim(A^{\#}, B^{\#}) = 1 \Leftrightarrow A^{\#} = B^{\#} \qquad \text{(Identität)} \tag{2.4}$$

$$sim(A^{\#}, B^{\#}) = sim(B^{\#}, A^{\#}) \qquad \text{(Symmetrie)} \tag{2.5}$$

Eng verwandt damit sind sogenannte Distanzmetriken, die eine Distanz zwischen zwei Elementen beschreiben. Neben der inversen Logik (die minimale Distanz entspricht einer maximalen Ähnlichkeit), müssen Distanzmetriken die sogenannte Dreiecksungleichung erfüllen (vgl. Sundaram 1996, S.6). Distanzmetriken sind zudem nicht auf den Wertebereich [0..1] beschränkt. Die Einhaltung der Dreiecksungleichung ist allerdings für den hier beschriebenen Anwendungsfall nicht notwendig und die explizite Einschränkung auf den Wertebereich [0..1] vereinfacht die Interoperabilität und den Vergleich zwischen verschiedenen

[1] Identität bezieht sich hier natürlich darauf, dass die Datensätze als gleich betrachtet werden, was ein Indiz aber keine Garantie für die Gleichheit der dazugehörigen Entitäten ist

Verfahren. Aus diesem Grund wird hier die Definition eines „Ähnlichkeitsmaßes“ bevorzugt. Distanzmetriken lassen sich jedoch meist problemlos in ein Ähnlichkeitsmaß überführen. Sofern die maximal mögliche Distanz zwischen 2 Elementen bekannt ist, beispielsweise über die Formel $1 - \frac{Distanz}{maximaleDistanz}$.

2.3 Ähnlichkeitmaße

Im Folgenden sollen einige gängige Ähnlichkeitsmaße beschrieben werden. Wenn Distanzmetriken beschrieben werden, so wird im Anschluss jeweils eine mögliche Umrechnungsformel angegeben, um diese in ein Ähnlichkeitsmaß zu überführen. Die meisten Ähnlichkeitsmaße sind über Zeichenketten definiert und müssen ggf. angepasst werden, um strukturierte Datensätze zu vergleichen. Am Beispiel der Kosinus-Ähnlichkeit wird dies im Kapitel „**Entwurf**“ beschrieben. Für andere Ähnlichkeitsmaße können zum Beispiel die verschiedenen Felder separat geprüft werden und anschließend ein (eventuell gewichteter) Mittelwert gebildet werden.

2.3.1 Edit-Distanz

Edit-Distanzmetriken beschreiben die günstigste Sequenz vordefinierter Operationen, die benötigt wird, um eine Zeichenkette in eine andere Zeichenkette zu überführen. Die erlaubten Operationen und deren Kosten unterscheiden sich je nach Variante.

Die Distanz zweier Zeichenketten ist definiert durch die minimalen Kosten, um eine Zeichenkette in die andere Zeichenkette zu überführen. Die Transformation wird dabei durch eine Sequenz vordefinierten Operationen realisiert und die Gesamtkosten aus der Summe der Einzelkosten für die benötigten Operationen berechnet. Die erlaubten Operationen und deren Kosten unterscheiden sich je nach Variante.

Am bekanntesten ist wohl die Levenshtein Distanz bei der Einfüge-, Lösch- und Ersetzungsoperationen definiert sind und die Kosten für alle Operationen konstant sind. (vgl. Levenshtein 1966)

Damerau (1964) beschreibt ein ähnliches Verfahren, bei dem allerdings auch die Vertauschung (Transposition) zweier benachbarter

Beispiel für die Levenshtein Distanz zweier Zeichenketten

```
Schmidt
|||||-| => Distanz: 1
Schmitt
```

Abbildung 2.1: Levenshtein Distanz zweier Zeichenkette

Zeichen als Operation definiert ist. Die so entstehende Distanzmetrik ist auch als Damerau-Levenshtein-Distanz bekannt.

Diese Art von Distanzmetrik eignet sich besonders gut für typographische (lexikalische) Fehler, wie Buchstabendreher oder versehentlich eingefügte beziehungsweise ausgelassene Zeichen, da die Distanz in diesen Fällen gering ist.

Phonetische Fehler („Meyer“ oder „Maier“) finden keine besondere Berücksichtigung im Algorithmus. Da die Struktur solcher Fehler aber oftmals typographischen Fehlern ähnelt (einzelne Zeichen ausgetauscht), werden diese dennoch vergleichsweise gut erkannt. Bessere Ergebnisse können durch eine Vorverarbeitung erzielt werden. Als Beispiel ist hier der Soundex Algorithmus zu nennen, der allerdings auf die englische Sprache abgestimmt ist.

Schlechte Ergebnisse erzielt man dagegen bei syntaktischen Abweichungen wie Feldvertauschungen („Müller, Stefan“ vs. „Stefan Müller“).

Die Umwandlung der Distanz in eine Ähnlichkeit im Bereich [0..1] kann zum Beispiel über die Berechnung $1 - \frac{Distanz}{max(\|A\|,\|B\|)}$ erfolgen.

2.3.2 Jaro–Winkler Ähnlichkeit

Die Jaro-Winkler Ähnlichkeit (vgl. Winkler 1990) ist ein weiteres Ähnlichkeitsmaß. Die Berechnung basiert auf der Methode von Jaro (1989) und wird von Winkler erweitert. Das Verfahren wird in der angegebenen Literatur nur unzureichend erklärt. Eine gute Beschreibung anhand des von Winkler verwendeten C-Codes mit vielen Beispielen findet sich unter Carpenter (2006).

2.3.3 Kosinus-Ähnlichkeit

Die Kosinus-Ähnlichkeit basiert auf der Idee, Zeichenketten als Vektoren in einem hochdimensionalem Vektorraum zu betrachten. Aus jeder Zeichenkette wird also zunächst ein Vektor gebildet, der die Zeichenkette repräsentiert. Die Umwandlung erfolgt so, dass ähnliche Zeichenketten auch eine ähnliche Vektor-Repräsentation haben. Wie genau die Umwandlung funktioniert, wird im nächsten Abschnitt beschrieben. Liegen die Zeichenketten in Vektordarstellung vor, so lässt sich die Ähnlichkeit zwischen zwei Zeichenketten als Winkel zwischen den dazugehörigen Vektoren definieren. Ähnliche Zeichenketten liegen auch im Vektorraum dicht beieinander, so dass der Winkel zwischen den Vektoren entsprechend gering ist. Unterschiedliche Zeichenketten liegen dagegen weiter auseinander, was in einem größeren Winkel resultiert.

Wenn definiert wird, dass alle Vektorkomponenten positiv sein müssen, so ist der Winkel zwischen zwei Vektoren maximal 90°. Der minimale Winkel, bei identischen oder sich nur in der Länge unterscheidenden Vektoren, beträgt 0°.

Der Winkel zwischen zwei Vektoren kann beispielsweise über das Skalarprodukt berechnet werden. Dazu wird zunächst der Kosinus berechnet:

$$\cos(\theta) = \frac{A \cdot B}{\|A\|\|B\|} = \frac{\sum_{i=1}^{n} A_i \times B_i}{\sqrt{\sum_{i=1}^{n} (A_i)^2} \times \sqrt{\sum_{i=1}^{n} (B_i)^2}} \tag{2.6}$$

Im zweiten Schritt kann nun aus dem Kosinus der Winkel ermittelt werden. Es macht allerdings Sinn, diesen letzten Schritt nicht durchzuführen, sondern den Kosinus direkt als eigentliches Ähnlichkeitsmaß zu benutzen. Während der Winkel zwischen 0° und 90° variieren kann, liegt der Kosinus bereits in dem von uns gewünschten Bereich von 0 bis 1. Auch die Bedeutung der Werte stimmt mit unserer Definition eines Ähnlichkeitsmaßes überein, denn ein Winkel von 0° entspricht einem Kosinus von 1 und damit der maximal definierten Ähnlichkeit. Kleinere Werte des Kosinus entsprechen dagegen größeren Winkeln; ein Wert von 0 entspricht der größtmöglichen Unähnlichkeit (Winkel von 90°). Wichtig ist, dass die Rangfolge der Ergebnisse gleich bleibt, egal ob Kosinus oder Winkel benutzt werden. Der Kosinus erfüllt also bereits alle Bedingungen eines Ähnlichkeitsmaßes.

Vektorraum-Transformation

Die größte Schwierigkeit bei diesem Ansatz besteht darin, einen Vektor zu finden, der eine Zeichenkette möglichst gut repräsentiert, so dass ähnliche Datensätze auch ähnliche Vektorrepräsentationen haben und damit der Winkel zwischen den Vektoren möglichst gering ist. Nur so erhält man später akkurate Ähnlichkeitswerte.

Eine Zeichenkette lässt sich in der Regel in mehrere Bestandteile (zum Beispiel Worte) zerlegen. Wenn davon ausgegangen wird, dass ähnliche Zeichenketten auch viele gemeinsame Worte enthalten, lässt sich dies für die Transformation in den Vektorraum nutzen. Betrachtet man jeden der Bestandteile („Terme“) als eine eigenständige Dimension und definiert die Vektoren dann so, dass die Komponenten den Wert 1 haben, wenn ein Term im dazugehörigen Datensatz vorkommt und ansonsten den Wert 0, erhält man ein sinnvolles Ähnlichkeitsmaß (siehe Beispiel in Abbildung 2.2).

Beispiel für die Kosinus-Ähnlichkeit

A =„Peter Müller“
B =„Peter Schmitt“
C =„Hans Schmitt“
T ={„Peter“, „Müller“, „Schmitt“, „Hans“}

	A	B	C
Peter	1	1	0
Müller	1	0	0
Schmitt	0	1	1
Hans	0	0	1

$cos(\theta)$	A^+	B^+	C^+
A	100%	50%	0%
B	50%	100%	50%
C	0%	50%	100%

Abbildung 2.2: Kosinus Ähnlichkeit mehrerer Zeichenketten

Das so resultierende Ähnlichkeitsmaß hat einige besondere Eigenschaften. So spielt zum Beispiel die Position eines Terms in der Zei-

chenkette keine Rolle. Es wird lediglich geprüft, ob ein Term in der Zeichenkette vorkommt (1) oder nicht (0). Dies bedeutet, dass auch die Reihenfolge der einzelnen Terme untereinander keine Bedeutung hat. Die Zeichenketten „"Peter Müller"“ und „"Müller, Peter"“ resultieren im gleichen Vektor[2] und haben eine Ähnlichkeit von 100%. Dies ist in vielen Fällen ein Vorteil gegenüber anderen Metriken wie der Levenshtein-Distanz, die in solchen Fällen nur eine sehr geringe Übereinstimmung bescheinigen würden.

Eine weitere Eigenschaft dieses Ansatzes ist, dass die resultierenden Vektoren in der Regel viele, oft tausende Dimensionen haben, da für jeden Term im Korpus eine eigene Dimension benötigt wird. Dies ist jedoch nicht wirklich tragisch, da der Vektor für eine einzelne Zeichenkette meistens nur in wenigen dieser Dimensionen einen von 0 verschiedenen Wert hat. Für die Berechnung der Kosinus-Ähnlichkeit spielen nur diese Komponenten eine Rolle.

Gewichtung

Die vorgestellte Vektorraumtransformation lässt sich noch weiter verfeinern, wenn unterschiedliche Termgewichte eingeführt werden. Statt also lediglich zwischen Vorkommen (1) und nicht-Vorkommen (0) zu unterscheiden, kann somit die Wichtigkeit (das „Gewicht“) für jeden Term innerhalb einer Zeichenkette festgelegt werden. Terme mit höherem Gewicht beeinflussen das Ergebnis der Kosinus-Ähnlichkeit stärker als Terme mit niedrigem Gewicht.

Ein mögliches Gewicht wäre beispielsweise, wie oft der Term in der Zeichenkette vorkommt. Terme die häufiger innerhalb einer Zeichenkette auftauchen, hätten damit einen größeren Einfluss auf das Gesamtergebnis. Ein weiterer zu betrachtender Faktor könnte die übliche Häufigkeit des Terms im Korpus, aus dem die Zeichenketten kommen, sein. Der Vorname „Peter“ ist zum Beispiel im Deutschen fast 1000 mal häufiger als der Vorname „Klaus-Günter“ (Balû 2009). Im ersten Fall ist die Wahrscheinlichkeit also deutlich höher, dass es sich bei einer Übereinstimmung lediglich um einen Zufall handelt wie im zweiten Fall.

Mehrere Faktoren lassen sich zu einem gemeinsamen Termgewicht kombinieren. Dabei unterscheidet man in der Regel zwischen lokalen und

[2]Kommas und andere Satzzeichen werden hier ignoriert

globalen Gewichtungsfaktoren. Ein lokaler Gewichtungsfaktor betrachtet lediglich die aktuelle Zeichenkette und wie der Term dort verwendet wird, während ein globaler Gewichtungsfaktor einen Term unabhängig von der jeweiligen Zeichenkette bewertet.

Die weit verbreitete tf-idf Gewichtung arbeitet nach diesem Prinzip. Als lokaler Gewichtungsfaktor dient hier die Häufigkeit des Terms in der Zeichenkette tf (Term Frequency) und als globaler Gewichtungsfaktor die Häufigkeit des Terms im Korpus insgesamt *idf* (Inverse Document Frequency) (vgl. Salton, Wong und Yang 1975).

Fazit

Da bei der Zerlegung der Terme nicht auf die Reihenfolge geachtet wird in dem die Terme auftreten, ist das Verfahren resistent gegen typische Feldvertauschungen, die bei der Levenshtein-Distanz ein Problem darstellten („Müller, Stefan“ vs. „Stefan Müller“).

Leider eignet sich das Verfahren in der hier vorgestellten Form nur sehr schlecht für typographische oder phonetische Fehler, da bei der Termzerlegung die ursprüngliche und die abweichende Version als zwei verschiedene Terme angesehen werden. In anderen Anwendungsgebieten wie z.B. dem Vergleich von Webseiten, spielt dies eine geringere Rolle, da wenige, fälschlicherweise als unterschiedlich erkannte Terme, durch die Übereinstimmung der restlichen Terme wieder ausgeglichen werden können. Adressdatensätze haben dagegen eine vergleichsweise geringe Länge, so dass die Ergebnisse nicht zufriedenstellend sind.

Um diesen Nachteil zu kompensieren, sind in der Literatur verschiedene Ansätze bekannt, die separat vorgestellt werden.

2.3.4 softTF-IDF-Ähnlichkeit

Eine Erweiterung der klassischen Kosinus-Ähnlichkeit stellt das softTF-IDF Ähnlichkeitsmaß dar. Bei der softTF-IDF Ähnlichkeit (vgl. Cohen, Ravikumar und Fienberg 2003) wird die Ähnlichkeit nicht nur durch Terme beeinflusst, die in beiden Datensätzen vorkommen, sondern auch durch Terme, die einem anderen Term im zweiten Datensatz ähneln. Dazu wird eine weitere Ähnlichkeitsfunktion sim', die sich gut für kurze Strings eignet (z.B. Levenshtein-Distanz oder Jaro-Winkler), benutzt, um die Terme der einzelnen Datensätze miteinander zu vergleichen.

Jeder Term wird dann mit dem Term aus dem anderen Datensatz gewichtet, zu dem er die höchste Ähnlichkeit hat, solange diese Ähnlichkeit über einem gewissen Schwellwert θ liegt.

Dieser Ansatz kombiniert damit die besten Eigenschaften von Kosinus-Ähnlichkeit (Erkennung von Feldvertauschungen) und Edit-Distanz (Erkennung von typographischen Fehlern) und liefert dementsprechend gute Ergebnisse. Die Berechnung der Ähnlichkeit zwischen den einzelnen Termen benötigt aber zusätzlichen Rechenaufwand.

Die Definition für zwei Zeichenketten A und B ist wie folgt:

Die Menge aller Terme in A, die ähnliche Terme in B haben, ist

$$Close(\theta, A, B) = \{a \in A | \exists b \in B \, sim'(a, b) > \theta\} \tag{2.7}$$

Die größte Ähnlichkeit eines Term aus A zu einem Term aus B ist

$$N(a, B) = max_{b \in B}(sim'(a, b)) \tag{2.8}$$

Die Termgewichte per TF-IDF Gewichtung sind

$$w(a, A) = Gewicht \, von \, a \, in \, A \tag{2.9}$$

Dann ist das softTF-IDF Gewicht

$$SSim(A, B) = \sum_{a \in Close(\theta, A, B)} w(a, A) w(a, B) N(a, B) \tag{2.10}$$

2.3.5 N-Gramm basierte Cosinus-Ähnlichkeit

Gravano u. a. (2003) beschreiben in ihrer Arbeit eine weitere Variante der Kosinus-Ähnlichkeit, die für die Termzerlegung keine Worte, sondern sogenannte N-Gramme (in der Regel Bi- oder Trigramme) benutzt.

N-Gramme sind alle zusammenhängende Zeichenketten der Länge N, die aus einem Eingabestring gebildet werden können. Die Trigramme ($N = 3$) für den Namen „Peter“ sind also die Zeichenketten „Pet“, „ete“ und „ter“. Typographische Fehler wirken sich in der Regel nur auf einen Teil der N-Gramme aus und beeinflussen dadurch den Ähnlichkeitswert weniger stark als bei der normalen Kosinus-Ähnlichkeit.

Da diese Art der Zerlegung ebenfalls sehr gute Ergebnisse zeigt und sich zudem, wie später beschrieben, sehr effizient implementieren lässt, bildet diese Variante die Basis für die hier vorgestellte Lösung und wird im nächsten Kapitel ausführlich beschrieben.

2.4 Partitionierungsstrategien

2.4.1 Paarweiser Vergleich

Die oben beschriebenen Ähnlichkeitsmaße definieren jeweils die Ähnlichkeit zwischen zwei Zeichenketten. Die einfachste Implementierung solcher Ähnlichkeitsmaße ist dementsprechend auch der paarweise Vergleich von jeweils zwei Datensätzen. Der Ansatz, auf diese Weise jeden Datensatz mit jedem anderem Datensatz zu vergleichen, stößt aber schnell an seine Grenzen, denn die Anzahl der benötigten Vergleiche betragt dann $\frac{n*n-1}{2}$ und die Laufzeit ist exponentiell ($O(n^2)$).

Eine einfache Losung besteht darin, nur Datensätze miteinander zu vergleichen, die zumindest eine grobe Ähnlichkeit zueinander haben. Jeder Datensatz wird dann anschließend nur mit den Datensätzen verglichen, die als „grob ähnlich“ markiert wurden. Es wird also faktisch ein zweites Ähnlichkeitsmaß definiert, das jedoch weniger auf Präzision, sondern mehr auf Geschwindigkeit und Trefferquote ausgelegt ist. Durch dieses zweite Ähnlichkeitsmaß werden die Datensätze in Gruppen von ähnlichen Datensätzen unterteilt. Innerhalb dieser Gruppen wird dann das eigentliche Ähnlichkeitsmaß angewendet. Je nach Partitionierungsstrategie variiert die Größe dieser Gruppen und sie können einander überlappen oder auch nicht.

Um einen Geschwindigkeitsvorteil zu erhalten, muss die Berechnung der „groben Ähnlichkeit“ natürlich deutlich schneller sein als der paarweise Vergleich durch das eigentliche Ähnlichkeitsmaß, das verwendet werden soll. Im Idealfall sollte zudem die Partitionierungsstrategie keinen Einfluss auf das Ergebnis haben. Alle Dubletten, die durch einen paarweisen Vergleich gefunden würden, sollten also auch nach der Partitionierung noch gefunden werden. Die in den Partitionen enthaltenen Datensatzpaare müssen dazu eine reine Obermenge der Dublettenpaare bilden, die durch das Ähnlichkeitsmaß festgelegt sind.

In der Praxis ist dies in der Regel nicht gegeben, da die meisten Partitionierungsstrategien sich aus Effizienzgründen deutlich von dem eigentlichen Ähnlichkeitsmaß unterscheiden. Dies wird zum großen Teil dadurch ausgeglichen, dass die Partitionen verhältnismäßig grob sind, so dass die meisten Dublettenpaare sich in der gleichen Partition befinden.

Eine geringe Beeinträchtigung ist in der Regel trotzdem gegeben (vgl. Abbildung 2.3).

Abbildung 2.3: Partitionierung und Dubletten

2.4.2 Blocking

Blocking-Techniken partitionieren die Eingabemenge in sich nicht überschneidende (disjunkte) Teilmengen. Dazu können verschiedene Strategien angewendet werden. Ein einfaches Beispiel ist die Verwendung eines einzelnen Attributes als Diskriminator. Für das Attribut „Postleitzahl“ würden also nur die Datensätze miteinander verglichen werden, die dieselbe Postleitzahl haben.

Eine einfache Verbesserung ist die Verwendung eines Präfixes, bei dem nur die ersten c Zeichen eines Attributes berücksichtigt werden. Dadurch könnte zum Beispiel die letzte Stelle der Postleitzahl ignoriert werden um Datensätze die sich nur in dieser letzten Stelle unterscheiden nicht im Vorhinein auszuschließen. Dieses Präfix kann dabei durchaus variabel gestaltet sein („Schm“ vs. „Z“).

Weitere Verbesserungen lassen sich erzielen, wenn bei mehreren Durchläufen verschiedene Attribute für das Blocking genutzt werden.

Die grundsätzliche Laufzeit dieser Methode hängt von der Anzahl der gebildeten Blöcke b ab. Die Anzahl der durchzuführenden Vergleiche ist $n \times \frac{n-b}{2b}$ (vgl. Draisbach und Naumann 2009). Wenn für die Sortierphase eine Laufzeit von $O(n \times log(n))$ angenommen wird, beträgt die

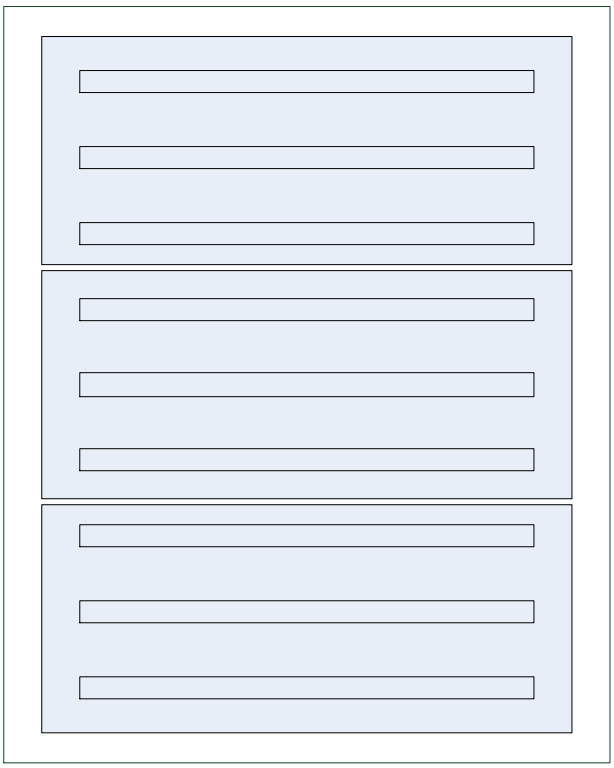


Abbildung 2.4: Blocking-Technik

Gesamtlaufzeit also $O(n(\frac{n}{2b} + log(n))$ [3]. Bei konstanter Blockgröße ($\frac{n}{b}$ konstant) kann somit eine Laufzeit von $O(n \times log(n))$ erreicht. Dabei sinkt allerdings bei steigendem n die Trefferquote, da die Datensätze auf mehr Blöcke aufgeteilt werden müssen. Ist dagegen die Anzahl der Blöcke konstant, ist die Gesamtlaufzeit quadratisch ($O(n^2)$).

2.4.3 Fenstertechnik

Diese Technik ist auch bekannt als „Sliding Window“-Technik oder „Sorted-Neighborhood“-Methode (vgl. Hernández und Stolfo 1995). Sie verwendet ein „Fenster“ fester Größe (w), das über die Datensätze bewegt wird. Ein Datensatz wird also immer mit den nächsten w Datensätzen verglichen, bevor das Fenster sich um einen Datensatz weiter bewegt. Alle Datensätze, die sich innerhalb dieses Fensters befinden, werden miteinander verglichen.

[3] Die Kosten für den Vergleich zweier Datensätze werden als konstant angenommen.

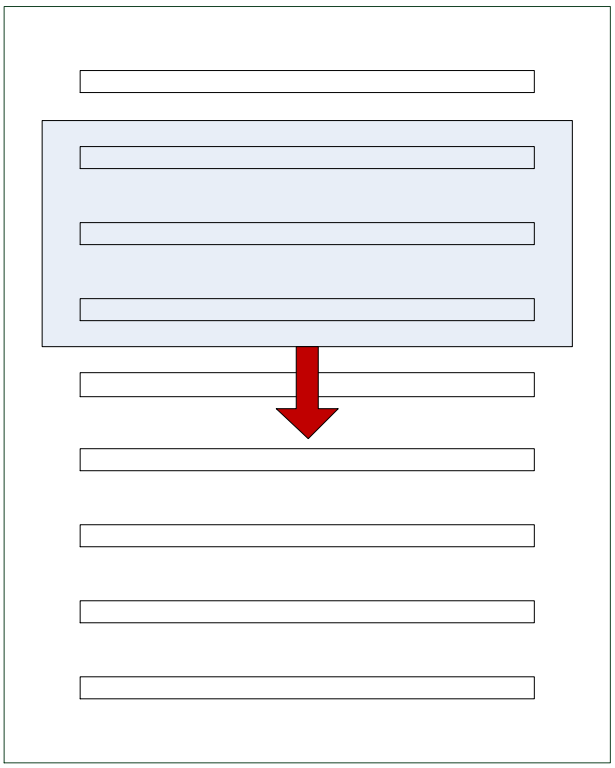


Abbildung 2.5: Fenstertechnik

Um möglichst ähnliche Datensätze miteinander zu vergleichen, werden die Daten nach bestimmten Attributen vorsortiert (z.B. Nachname). Dies stellt auch den größten Nachteil dieser Technik dar, denn wenn sich zwei Dubletten genau in dem Attribut unterscheiden nach dem sortiert wurde, so befinden sie sich möglicherweise nicht im selben Fenster und werden nicht als Dubletten erkannt. Selbst bei gleichem Attributwert kann es passieren, das Dubletten übersehen werden, wenn die Anzahl der Datensätze mit gleichem Attributwert größer als das Fenster ist (Beispiel: 2000 „Müller“ aber $w = 1000$).

Abhilfe schafft es, die Datensätze mehrmals auf Dubletten zu überprüfen und dabei jedes Mal nach unterschiedlichen Schlüsseln zu sortieren, da dadurch die Wahrscheinlichkeit deutlich verringert wird, dass Dubletten übersehen werden. Hernández und Stolfo (1998) zeigen, dass diese Anpassung deutlich mehr Auswirkung auf die Effektivität der Dublettenerkennung hat, als eine Erhöhung der Fenstergröße. So hatte eine Fenstergröße von mehr als 30 Datensätzen nur noch marginale Auswirkungen auf die Erkennungsrate, während drei Durchläufen mit

verschiedenen Schlüsseln die Erkennungsrate von etwa 55% auf ungefähr 90% erhöhte.

Ein weiterer Nachteil dieser Methode ist, dass vom Benutzer eine Auswahl von geeigneten Schlüsseln getroffen werden muss, was Fachwissen über die verwendete Methodik und eine gewisse Einarbeitungszeit erfordert.

Die Gesamtlaufzeit der Dublettensuche mit Fenstertechnik bei n Datensätzen setzt sich zusammen aus der Sortierphase (bestenfalls $O(n \times log(n))$) und der Vergleichsphase ($O(wn)$) (vgl. Leser und Naumann 2007, S.341). Die gesamte asymptotische Laufzeit ist damit O(n(w+log(n)).

Untersuchungen von Draisbach und Naumann (2009) deuten darauf hin, dass die Fenstertechnik bei Auswahl der gleichen Schlüssel deutlich effektiver als entsprechende Blocking-Techniken sind. Sie stellen außerdem eine Möglichkeit vor, beide Techniken zu kombinieren.

2.4.4 Moderne Verfahren

Während traditionelle Partitionierungsstrategien von der Ähnlichkeitsfunktion ausgehen und diese als paarweisen Vergleich implementieren, wird im Bereich der Volltext-Suchmaschinen in der Regel ein anderer Ansatz gewählt. Da hier von Anfang an die Effizienz eine große Rolle spielt, wird versucht, bekannte Techniken aus dem Datenbankbereich (z.B. Indices) so zu erweitern, dass unscharfe Abfragen gemacht werden können, also keine Gleichheit, sondern lediglich eine bestimmte Ähnlichkeit gefordert wird.

Sogenannte „Similarity Joins“ stellen dabei das theoretische Fundament dar, das auch für die Suche nach ähnlichen Datensätzen verwendet werden kann. Wie bei einem herkömmlichen Join der relationalen Algebra, werden bei einem Similarity Join zwei Datenmengen miteinander verknüpft. Bei einem Similarity Join wird zusätzlich noch ein Schwellwert übergeben mit dem alle Datensatzpaare zurückgegeben werden, deren Ähnlichkeit über dem Schwellwert liegt.

Große Fortschritte wurden darin vor allem in den letzten Jahren gemacht. Einer der ersten Ansätze, die klassische relationale Algebra um Ähnlichkeitsprädikate zu erweitern findet sich bei Sarawagi und Kirpal 2004. Aus dem Forschungszentrum von Google stammen Vorschläge, wie

die Implementierung dieser Ansätze noch effizienter gestaltet werden kann (vgl. Bayardo, Ma und Srikant 2007).

Allen diesen Verfahren gemeinsam ist, dass sie ein vollständiges Ergebnis in Bezug auf die Ähnlichkeitsfunktion zurückliefern. Eine möglicherweise fehlerbehafteter expliziter Partitionierungsschritt fällt also weg. Dank des intelligenten Einsatzes von Indices kann trotzdem eine hohe Effizienz erreicht werden, die in vielen Fällen sogar die klassischen Verfahren übertrifft. Dazu werden diverse Optimierungen verwendet, die rein mathematisch versuchen potenzielle Kandidaten im Vorhinein auszuschließen, wenn deren Ähnlichkeit zueinander den vorgegebenen Schwellwert nicht erreichen kann.

Viele der Veröffentlichungen beziehen sich auf die Erkennung von Duplikaten von Webseiten oder in anderen Dokumentenbeständen. Die Anwendung in klassischen Adressdatenbanken scheint weniger erforscht zu sein. Das liegt vielleicht auch daran, dass diese mit klassischen Methoden noch zu bewältigen sind und deren Implementierung deutlich einfacher ist. Die im nächsten Kapitel vorgestellte Lösung soll zeigen, wie modernen Verfahren sich auf die Erkennung von Dubletten in Adressbeständen oder ähnlichen Datenbanken anwenden lassen.

2.5 Verfügbare Produkte

Sucht man in den bekannten Suchmaschinen nach Software, die auf die Dublettenerkennung bzw. Bereinigung spezialisiert ist, so finden sich erstaunlich wenig Angebote. Das liegt vor allem daran, dass eine Dublettensuche möglichst gut in die bestehende Infrastruktur (also z.B. die CRM-Software) integriert werden muss. Für viele CRM-Produkte gibt es daher integrierte Maßnahmen um Dubletten zu erkennen oder im Vorfeld zu verhindern. Darauf soll im zweiten Teil dieses Abschnitts eingegangen werden, während zunächst einige eigenständige Produkte beschrieben werden.

2.5.1 BatchDeduplicator

Der BatchDeduplicator von Dipl. Inf. Thomas Hainke (vgl. *Batch-Deduplicator 4.04 - Die Software mit der Datenqualität planbar wird*) ermöglicht nach eigenen Angaben eine automatische, Zeitplan-gesteuerte

Dublettensuche in diversen Datenquellen (z.B. Access, Excel, CSV Dateien, sowie verschiedenen DBMS).

Als Abgleichskriterium können – neben einigen vorgefertigten Profilen – beliebige Felder gewählt werden, wobei auch ein unscharfer Abgleich möglich ist, bei dem Dubletten trotz Rechtschreibfehler oder vertauschten Feldern erkannt werden.

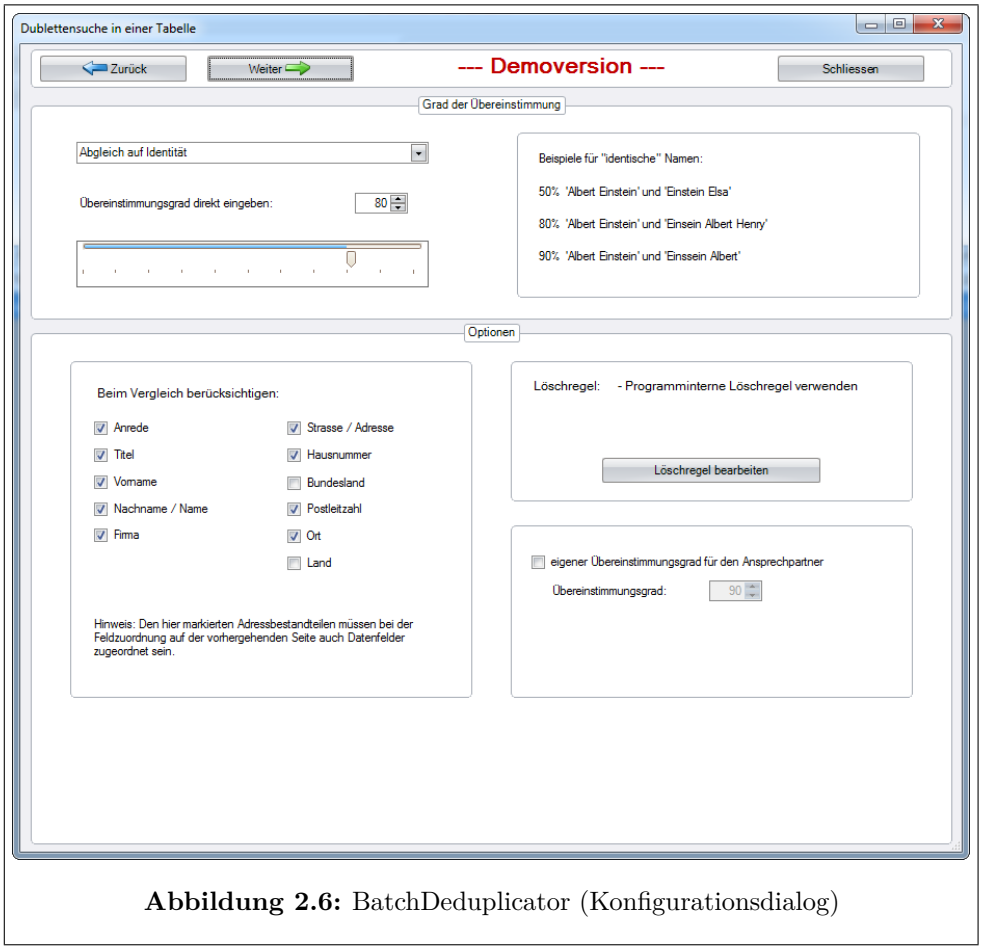


Abbildung 2.6: BatchDeduplicator (Konfigurationsdialog)

Nach eigenen Angaben kann das Programm auch mit großen Datenmengen umgehen, wobei die Bearbeitung von 5,5 Millionen Datensätzen in unter einer Stunde beworben wird. Die genauen Testparameter dazu sind allerdings nicht bekannt. Eigene Tests mit einem Datenbestand von etwa 65.000 Adressen wurden innerhalb von wenigen Sekunden durchgeführt, allerdings nur solange man das vorgefertigte Profil für Adressen benutzt. Bei der freien Definition von Feldern brauchte das Programm deutlich länger (etwa 6 Minuten), wobei nicht auszuschließen ist, dass dies auf fehlerhafte Bedienung des Programmes zurückzuführen ist.

Die Anzahl der gefundenen Dubletten war bei der Verwendung der Standardeinstellung vergleichsweise gering, die Qualität der Ergebnisse aber durchweg gut und es wurden kaum Datensätze fälschlicherweise als Dubletten erkannt. Nach der Ausführung lassen sich die Ergebnisse automatisch in verschiedenen Formaten exportieren. Über die verwendeten Algorithmen ist nichts bekannt.

Die Software ist auf allen gängigen Windows Versionen lauffähig und wird für 498€ (ohne Mwst., Stand: Februar 2012) angeboten.

2.5.2 FuzzyDupes

FuzzyDupes ist eine Software von Kroll-Software, die ebenfalls der Dublettenerkennung dient (vgl. *FuzzyDupes 2011 - Unscharfe Dublettensuche in Datenbanken*). Auch hier werden die gängigen Datenquellen wie CSV, Excel, Access sowie beliebige DBMS (über ODBC) unterstützt. Im Gegensatz zum BatchDeduplicator macht der Hersteller einige Angaben zu dem verwendeten Algorithmus. So erfährt man, dass dieser aus einem zweistufigen Verfahren besteht, bei dem zunächst Partitionen anhand eines „Trigram-Hashindex“ gebildet werden und anschließend der eigentliche Vergleich anhand eines selbst entwickelten Verfahrens durchgeführt wird, das „besser als alle bekannten Verfahren Permutationen berücksichtigen kann“. Es wird darauf hingewiesen, dass mathematisch beweisbar alle Ähnlichkeiten erkannt werden (sofern man das Verfahren kennt). Was genau das bedeuten soll, wenn man davon ausgeht, dass es keine einheitliche Definition von Ähnlichkeit gibt, wird leider nicht erwähnt. Die genauen Details des Algorithmus bleiben unbekannt.

Bei der Ausführung des Programmes müssen die für die Partitionierung zu verwendenden Attribute speziell vom Benutzer markiert werden. Auch werden eine Vielzahl von weiteren Konfigurationsparametern verlangt (z.B. „Schwellenwert Cluster“) mit denen wohl viele Benutzer überfordert sind. Der verlinkte Hilfetext ist recht kurz gehalten und erklärt kaum, welche Auswirkungen die einzelnen Parameter haben. Meistens wird darin geraten, die Standardeinstellungen zu benutzen, die bei eigenen Tests tatsächlich zu guten Ergebnissen führten. Die Standardeinstellung von 90% Übereinstimmung führte vereinzelt noch zu falsch erkannten Duplikaten, eine Erhöhung auf etwa 93% beseitigte in diesem konkreten Fall das Problem zuverlässig, wobei man in Kauf

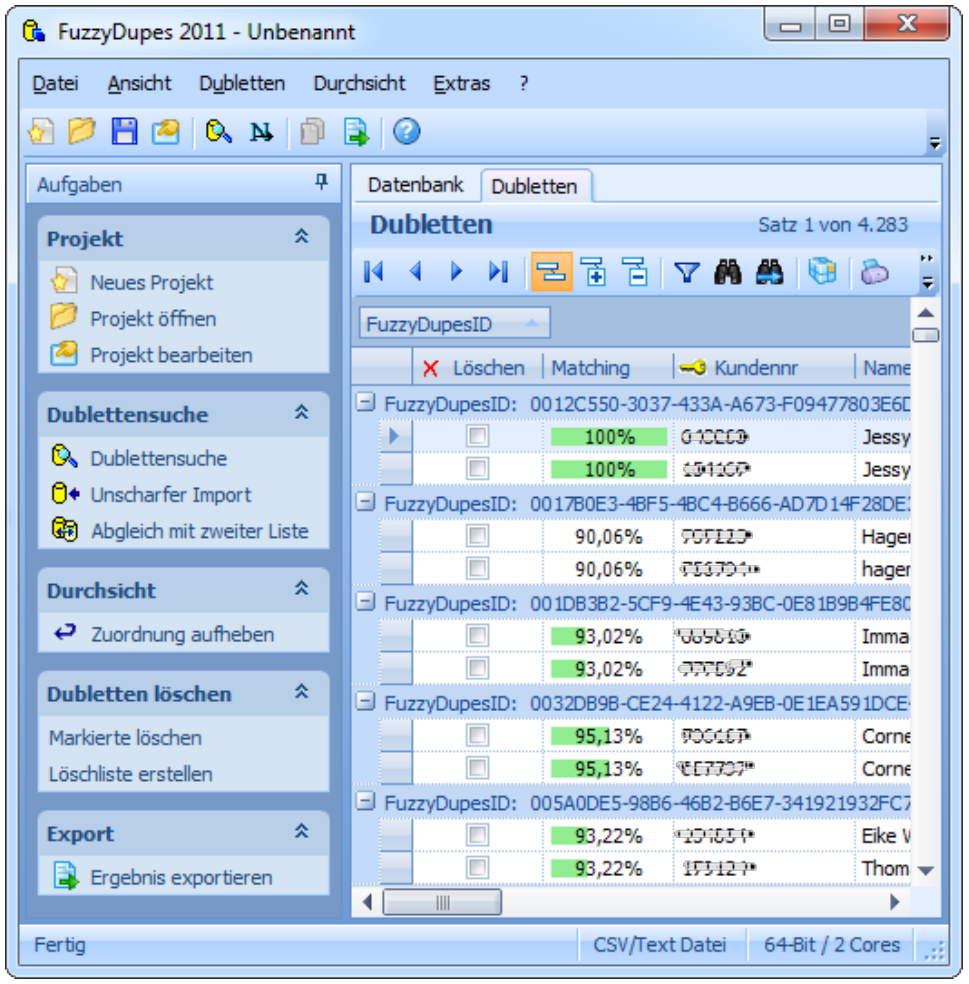


Abbildung 2.7: FuzzyDupes: Startfenster mit Ergebnissen

nehmen musste, dass dann einige tatsächliche Dubletten nicht mehr erkannt wurden.

Der Hersteller selber macht keine Angaben darüber, bis zu welcher Datenbankgröße die Dublettenerkennung geeignet ist. Der Test mit 65.000 Adressen wurde bei Standardeinstellungen in etwa 30 Sekunden fertiggestellt, der gleiche Test mit etwa 300.000 Adressen benötigte bereits etwas mehr als 10 Minuten, wobei sich dies durch eine Verringerung des Cluster Schwellenwerts laut Autor noch beschleunigen lässt.

Während der Erkennung wird – neben dem Fortschritt – eine Vielzahl von Statistiken ausgegeben, die aber ohne Kenntnisse des dahinter liegenden Algorithmus nicht gedeutet werden können und daher für

die meisten Benutzer wohl eher verwirrend sein dürften (siehe Abbildung 2.8).

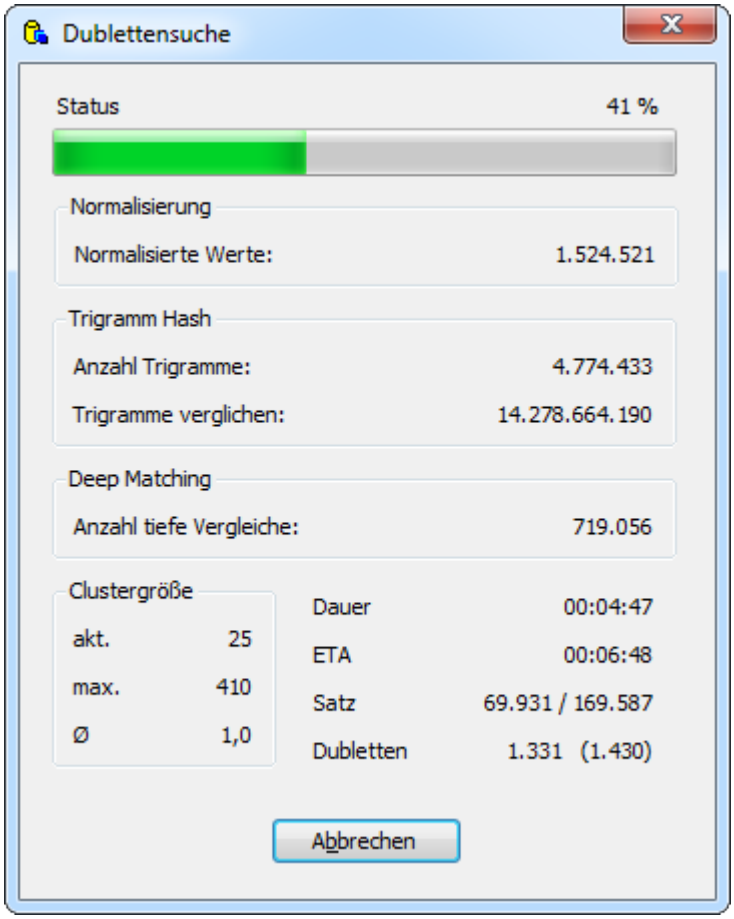


Abbildung 2.8: FuzzyDupes: Fortschritt der Dublettenerkennung

Auch hier lassen sich die Ergebnisse exportieren oder direkt in der Datenquelle löschen. FuzzyDupes lässt sich (Stand: Februar 2011) für 349€ (ohne Mwst.) erwerben. Interessant ist auch die Option für 219€ eine .NET bzw. COM-Bibliothek zu erwerben, mit dem die Dublettenerkennung in eigene Anwendungen integriert werden kann.

2.5.3 CRM Systeme

Für die gängigen CRM Systeme (Oracle/Siebel, SAP, Microsoft Dynamics, ...) gibt es entweder integrierte oder externe Lösungen um eine Dublettenerkennung direkt innerhalb der Anwendung zu ermöglichen.

Diese erkennen Dubletten in der Regel schon vor dem Speichern der betroffenen Datensätzen und warnen den Benutzer, wenn nötig.

In Microsoft Dynamics CRM lassen sich in der integrierten Dublettenerkennung Regeln erstellen, die – bei der Übereinstimmung von bestimmten Feldern – Datensätze als Dubletten markieren. Diese Regeln sind relativ primitiv und erlauben lediglich die exakte Übereinstimmung der Felder oder eine Übereinstimmung der ersten oder letzten n Zeichen im Feld als Kriterium. Es gibt kein Konzept einer prozentualen Übereinstimmung. Dies führt zum Teil zu kreativen Ideen wie die Erkennungsleistung verbessert werden kann (vgl. De Coninck 2009).

Für SAP, Siebel und PeopleSoft CRM Systeme gibt es unter anderem Lösungen von „Firstlogic Solutions“, die neben der Dublettenerkennung auch weitere Data Cleansing Aufgaben unter dem Namen „Data Quality Managment“ anbieten. Genauere Details zu den verwendeten Verfahren sind nicht bekannt (vgl. *Firstlogic Solutions - Data Quality. Delivered.*).

3 Entwurf

3.1 Anforderungen

Viele der vorhandenen Lösungen zur Dubletten-Erkennung wurden für ein bestimmtes Produkt entworfen oder sind bereits in dieses integriert. Eigenständige Lösungen sind dagegen oft schwer zu konfigurieren und lassen sich nur unzureichend automatisieren oder in andere Produkte integrieren.

Oft wird zudem auf klassische Algorithmen zur Dublettenerkennung zurückgegriffen, die zwar gute Dienste leisten, aber aus heutiger Sicht nicht mehr aktuell sind. Moderne Algorithmen können die Leistung, bei gleichzeitiger Erhöhung der Erkennungsleistung, steigern.

Die hier vorgestellte Lösung soll eine Dubletten-Erkennung für mittlere bis große Datenbestände realisieren. Dabei soll die einfache Integration und Konfiguration der Dublettenerkennung im Vordergrund stehen, so dass sich die Lösung leicht für verschiedene Kunden anpassen lässt. Mit Hilfe moderner Algorithmen soll darüber hinaus eine hohe Effizienz bei gleichzeitig guter Erkennungsleistung erreicht werden.

3.2 Übersicht

Der für die Dublettenerkennung verwendete Algorithmus besteht aus mehreren, aufeinander aufbauenden Schritten (vgl. Abbildung 3.1).

Der Ablauf beginnt mit der **Vorverarbeitung**, bei der die Datensätze durch definierbare Regeln normalisiert werden.

Im Anschluss werden einzelne Datensätze, die für die Dublettenerkennung nicht geeignet sind, herausgefiltert (**Filterung**).

Zur eigentlichen Berechnung wird die Kosinus-Ähnlichkeit mit Trigrammen eingesetzt, die speziell erweitert und optimiert wurde. Dazu müssen die verbleibenden Datensätze zunächst in Trigramme zerlegt werden. Die Trigramme werden dann zusammen mit den dazugehören-

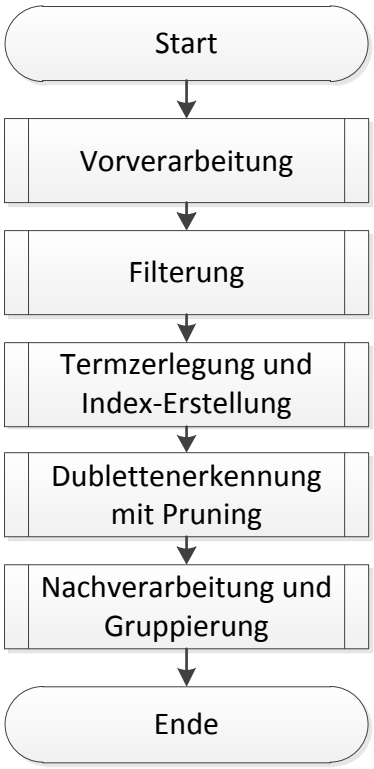


Abbildung 3.1: Ablauf der Dublettenerkennung

den Datensätzen in einem inversen Index gespeichert (**Termzerlegung und Index-Erstellung**).

Nun kann die eigentliche Dublettenerkennung durch Suche innerhalb der Index-Strukturen erfolgen. Das verwendete „Pruning“-Verfahren sorgt dafür, dass viele Kandidaten bereits frühzeitig ausgeschlossen werden können und der Algorithmus dadurch beschleunigt wird (**Dublettenerkennung mit Pruning**).

Als finaler Schritt werden die Ergebnisse für den Benutzer aufbereitet in dem z.B. einzelne Dublettenpaare zu Gruppen zusammengefasst werden (**Nachverarbeitung und Gruppierung**).

Die einzelnen Schritte werden in den folgenden Abschnitten detailliert beschrieben.

3.3 Vorverarbeitung

Die Eingabedaten werden vor der eigentlichen Dublettenerkennung zunächst anhand eines definierbaren Regelsatzes normalisiert. Ein typischer Regelsatz beinhaltet die Entfernung von Punktierung aus einem Feld und die Vereinheitlichung von Groß- und Kleinschreibung. Mehrere Felder können zudem zum Zwecke der Dublettenerkennung zu einem neuen logischen Feld zusammengefasst werden. Dadurch werden Ähnlichkeiten zwischen Datensätzen auch feldübergreifend erkannt. Typischer Anwendungsfall sind Feldvertauschungen (Name vs. Vorname).

Mathematisch gesehen entspricht dies einer Abbildung der Quelldatenmenge Ψ_{org} auf eine neue Menge Ψ_{trans}. Jedem Datensatz $D^{\#}$ aus Ψ_{org} wird ein neuer Datensatz aus der Menge Ψ_{trans} zugeordnet, der anhand der Regeln normalisiert wurde:

$$\Psi_{trans} = transform[\Psi_{org}] \tag{3.1}$$

$$= \{transform(D^{\#}) | D^{\#} \in \Psi_{org}\} \tag{3.2}$$

Beispiel: Normalisierung eines Datensatzes

Regelsatz: Großschreibung, nur Zahlen und Buchstaben

$D^{\#} \in \Psi_{org}$ ={„Peter“, „Müller“, „Hauptstr. 123“}

$\Downarrow_{transform}$

$D^{\#}_{trans} \in \Psi_{trans}$ ={„PETER“, „MÜLLER“, „HAUPTSTR 123“}

Abbildung 3.2: Normalisierung eines Datensatzes

3.4 Filterung

In diesem Schritt werden Datensätze, die zu viele Felder mit leeren Werten (*NULL*) aufweisen, rausgefiltert. Solche Datensätze sind oft

nur schwer mit anderen Datensätzen vergleichbar, weil für eine verlässliche Aussage über Ähnlichkeit oder Unähnlichkeit nicht genügend Informationen bereitstehen. Neben der reinen Ähnlichkeit, muss also auch die Qualität der Beurteilung berücksichtigt werden, was bei der reinen Kosinus-Ähnlichkeit nicht gegeben ist. Diese Qualität ist abhängig davon wie hoch der Anteil an leeren Feldern im Datensatz ist (seine „Vollständigkeit“). Allerdings ist nicht jedes Feld im Datensatz gleich wichtig. Das Fehlen der Angabe „Geschlecht“ mag zum Beispiel weniger relevant sein, wie das Fehlen der Angabe „Name“. Um diese Tatsache abzubilden kann jedem Feld f ein Feldgewicht w_f zugeordnet werden[1], das ausdrückt wie wichtig dieses Feld im Vergleich zu anderen Feldern ist. Diese Feldgewichte dienen nicht nur der Beurteilung der Vollständigkeit sondern werden auch in die Berechnung der Ähnlichkeit einbezogen, wie später erläutert wird.

Die Vollständigkeit c eines Datensatzes $D^{\#}$ kann dann als Quotient der Feldgewichte deren Felder $NULL$ sind und der Summe der Feldgewichte insgesamt definiert werden.

$$c_{D^{\#}} = \frac{\sum_{f \in F} w_f [D_f^{\#} \neq NULL]}{\sum_{f \in F} w_f} \tag{3.3}$$

Werden zwei Datensätze miteinander verglichen, so sinkt die Verlässlichkeit der Angabe, dass es sich bei den Datensätzen um eine Dublette handelt, wenn mindestens einer der beiden Datensätze eine geringer Vollständigkeit hat – selbst dann, wenn die Ähnlichkeit sehr hoch ist. Diese „Reliabilität“ wird im Algorithmus als zusätzlicher Maßstab[2] berechnet und zurückgegeben und ist als die minimale Vollständigkeit der beteiligten Datensätze definiert.

[1] Statt die Feldgewichte direkt vom Benutzer zuordnen zu lassen, kann dies zum Beispiel auch über Heuristiken erfolgen, die den Informationsgehalt eines Feldes beurteilen.

[2] Neben dem Ähnlichkeitswert.

Beispiel: Ähnlichkeit und Reliabilität bei *NULL*-Werten

Vorname (25%*)	**Name (25%*)**	**Straße (50%*)**	**Ähnlichkeit**	**Reliabilität**
PETER	*NULL*	*NULL*	100%	25%
PETER	*NULL*	*NULL*		
PETER	MÜLLER	*NULL*	100%	50%
PETER	MÜLLER	*NULL*		
PETER	MÜLLER	HAUPTSTR 123	100%	100%
PETER	MÜLLER	HAUPTSTR 123		

*Feldgewichte

Tabelle 3.1: Ähnlichkeit und Reliabilität bei *NULL*-Werten

Kandidatenpaare mit geringer Reliabilität können vom Benutzer anhand eines Schwellwertes ν ausgeschlossen werden. Da die Reliabilität immer der minimalen Vollständigkeit der beteiligten Datensätze entspricht, müssen Datensätze mit einer Vollständigkeit $c < \nu$ nicht bei der Dublettenerkennung berücksichtigt werden und können im Vorhinein aussortiert werden.

$$\Psi = \sigma_{c \geq \nu}(\Psi_{trans}) \tag{3.4}$$

$$= \{D^{\#} \in \Psi_{trans} | c_{D^{\#}} > \nu\} \tag{3.5}$$

Die transformierte und gefilterte Datensatzmenge Ψ bildet die Eingabe für alle weiteren Berechnungsschritte des Algorithmus.

3.5 Datensatzzerlegung

Zur Überführung der Datensätze in einen Vektorraum zum Zwecke der Kosinus-Berechnung, müssen zunächst die Dimensionen des Vektorraums definiert werden. Dazu werden die Datensätze in Trigramme zerlegt. Ein Trigramm ist eine Sequenz von drei aufeinanderfolgenden Zeichen im Datensatz[3]. Die Verwendung von Trigrammen statt Wör-

[3]Trigramme werden in dieser Implementierung nicht über Wortgrenzen hinweg gebildet. Wenn ein Wort aus weniger als drei Zeichen besteht, so wird stattdessen

tern dient dabei der besseren Erkennungsrate bei typographischen oder lexikalischen Unterschieden.

Um Trigramme, die aus unterschiedlichen Feldern stammen, zu unterscheiden, werden diese zusätzlich über einen Feldindex qualifiziert. Das Tupel $(Trigramm, Feldindex)$ wird dann als Token (t) bezeichnet. Tokens mit gleichen Trigrammen aus unterschiedlichen Feldern gelten demnach nicht als äquivalent. Würde diese Einteilung aufgehoben, so würden Daten aus unzusammenhängenden Feldern miteinander verglichen werden. Die Postleitzahl „51643“ hätte damit eine hohe Übereinstimmung mit der Telefonnummer „0178-**51643**-75“ und die dazugehörigen Datensätze würden möglicherweise zu unrecht als ähnlich eingestuft. Wo es dagegen Sinn macht auch feldübergreifend zu vergleichen, kann man dies über entsprechende Vorverarbeitungsschritte lösen, in dem Felder, bei denen eine vergleichsweise hohe Verwechslungsgefahr besteht (z.B. „Name“ vs. „Vorname“), zusammengefasst werden. Da die Unterscheidung verschiedener Felder bereits über die einzelnen Token gegeben ist, muss die Liste der Tokens eines Datensatzes nicht noch einmal nach Feldern separiert werden, was die späteren Berechnungsschritte vereinfacht.

Die als Tokenliste serialisierte Form eines Datensatzes soll im Folgenden auch als Dokument bezeichnet werden, um sie von der ursprünglichen, feldbasierten Repräsentation abzugrenzen. Das Dokument D enthält dann die Liste aller Token t des Datensatzes $D^{\#}$, die Tokenmenge T ist die Gesamtmenge aller Token im Dokument-Korpus Λ und *tokenize* ist eine Funktion die einen Datensatz auf seine Tokenliste abbildet:

$$D = tokenize(D^{\#}) \tag{3.6}$$

$$\Lambda = tokenize[\Psi] \tag{3.7}$$

$$T = \bigcup_{D \in \Lambda} D \tag{3.8}$$

das gesamte Wort als Uni- oder Bigramm aufgenommen und im weiteren Verlauf wie ein reguläres Trigramm behandelt

3.6 Gewichtung

Zur Gewichtung der einzelnen Token wird eine Kombination aus lokalen und globalen Gewichtungsfaktoren benutzt. Während lokale Gewichtungsfaktoren die Verwendung des Tokens im Dokument betrachten, wird bei globalen Gewichtungsfaktoren ein Token unabhängig von einem einzelnen Dokument betrachtet (vgl. Abschnitt 2.3.3).

Das Gesamtgewicht w_{tot} eines Tokens t im Dokument D kann damit in einen lokalen Anteil w_l und einen globalen Anteil w_g aufgeteilt werden:

$$\underbrace{w_{tot}(t,D)}_{Gesamtgewicht} = \underbrace{w_g(t)}_{globalerAnteil} \times \underbrace{w_l(t,D)}_{lokalerAnteil} \tag{3.9}$$

3.6.1 Globale Gewichtungsfaktoren

Der globale Anteil besteht aus der inversen Dokumenthäufigkeit idf_t des Tokens t und einem Feldgewicht w_f.

$$w_g(t) = \underbrace{idf_t}_{inverseDokumenthäufigkeit} \times \underbrace{w_f}_{Feldgewicht} \tag{3.10}$$

$$\tag{3.11}$$

Inverse Dokumenthäufigkeit

Die Idee hinter der inversen Dokumenthäufigkeit ist, dass Token die in vielen unterschiedlichen Dokumenten vorkommen eine geringere Bedeutung haben wie Token die eher selten vorkommen. Das Token „STR“ kommt zum Beispiel (als Kürzel für „Straße“) besonders häufig in Adressfeldern vor. Die Übereinstimmung eines solchen Tokens zwischen zwei Datensätzen hat also keine besondere Aussagekraft und sollte dementsprechend ein geringes Gewicht haben. Kommt dagegen ein sehr selten vorkommendes Token in zwei unterschiedlichen Datensätzen vor, kann dies ein Indiz dafür sein, dass es sich bei den beiden Datensätzen um Dubletten handelt. Solche Token sollten daher ein größeres Gewicht haben.

Die inverse Dokumenthäufigkeit für ein Token t wird definiert als der Quotient aus der Anzahl aller Datensätze und der Anzahl der Datensätzen die das Token t enthalten. Da der Quotient alleine aus Erfahrung

zu sehr großen Unterschieden in der Gewichtung führen würde, wird stattdessen der Logarithmus dieser Zahl verwendet. Zusätzliche Additionen von 1 sorgen dafür, dass auch in Grenzfällen der Quotient keinen negativen Nenner hat und das Ergebnis in jedem Fall positiv ist:

$$idf_t = 1 + \ln\left(\frac{\overbrace{\|\Psi\|}^{\text{Anzahl aller Datensätze}}}{\underbrace{\|\Psi_t\|}_{\substack{\text{Anzahl der Datensätze,}\\ \text{die Token t enthalten}}} + 1}\right) \tag{3.12}$$

Beispiel: Inverse Dokumenthäufigkeiten

Token	Häufigkeit*	Inverse Dokumenthäufigkeit
PET	211	4,85
ETE	290	4,54
TER	438	4,13
MÜL	182	5,00
ÜLL	186	4,98
LLE	350	4,35
LER	650	3,73
HAU	212	4,85
AUP	58	6,13
UPT	53	6,22
PTS	16	7,38
TST	186	4,98
STR	6606	1,41
123	3	8,82

* Häufigkeiten aus 10000 statistisch generierten Adressen (vgl. Abschnitt 5.1)

Tabelle 3.2: Inverse Dokumenthäufigkeiten

Tabelle 3.2 zeigt die Tokens der Beispiel-Adresse „Peter Müller, Hauptstr. 123“ mit ihrer Häufigkeit im Dokumentkorpus und die inverse Dokumenthäufigkeit.

In dem Beispiel lässt sich gut sehen, dass besonders häufig vorkommende Terme wie das „STR“ (Straße) aus dem Adressfeld gegenüber anderen Token deutlich geringer gewichtet werden. Sehr selten vorkommende Terme, wie die Hausnummer „123“, die insgesamt nur drei Mal in den 10000 Adressen vorkommt, werden dagegen deutlich höher gewichtet.

Feldgewicht

Verschiedene Felder aus dem ursprünglichen Datensatz können eine unterschiedlich hohe Relevanz für die Beurteilung des Datensatzes als Dublette haben. Ein Feld, das lediglich das Geschlecht einer Person als weiblich „w“ oder männlich „m“ festhält, hat zum Beispiel eine geringere Selektivität als das Feld „Postleitzahl“ und sollte daher weniger Gewicht haben. Im Abschnitt 3.4 wurde für die Relevanz eines Datenfeldes bereits ein Feldgewicht w_f eingeführt, welches auch hier in der Beurteilung mit einbezogen wird. Dieses Feldgewicht muss vor Aufruf des Algorithmus konfiguriert werden. Dies kann entweder durch den Benutzer geschehen oder auch automatisch durch Heuristiken festgelegt werden, die den Informationsgehalt eines Feldes abzuschätzen versuchen.

3.6.2 Lokale Gewichtungsfaktoren

Als lokaler Gewichtungsfaktor wird häufig die Anzahl der Vorkommen tf eines Tokens im Datensatz benutzt. Damit werden Token stärker gewichtet, wenn sie im Datensatz häufiger vorkommen. Dies bietet sich auch für die Dublettenerkennung an. Die Annahme dabei ist, dass häufig vorkommende Token besonders relevant sind.

Als weiteres Kriterium muss sichergestellt werden, dass Felder, die Daten mit geringer Länge enthalten (z.B. „Postleitzahl“), gegenüber anderen, längeren Feldern („Anschrift“) nicht benachteiligt werden. Längere Felder haben nicht zwangsläufig einen höheren Informationsgehalt, würden aber durch eine höhere Anzahl an Token auch einen höheren Einfluss auf das Ergebnis haben.

Da die Wichtigkeit eines bestimmten Feldes bereits über das Feldgewicht als globalen Faktor berücksichtigt wird, sollte die Länge der darin enthaltenden Daten keine Rolle mehr spielen. Token in langen Feldern müssen daher abgewertet werden, damit das lokale Gesamtgewicht des

Feldes konstant bleibt. Dies erreicht man in dem das lokale Gewicht eines Tokens durch die Länge des dazugehörigen Feldes geteilt wird. Die Länge $fsize$ eines Feldes f ist dabei definiert als die Anzahl der Token im Dokument D die zu diesem Feld gehören.

Als lokales Gesamtgewicht ergibt sich damit:

$$w_l(t) = \frac{\overbrace{tf(D^{\#}, t)}^{\text{Häufigkeit von } t \text{ in } D}}{\underbrace{fsize}_{\substack{\text{Anzahl der Token in } D, \\ \text{die aus Feld } f \text{ stammen}}}} \tag{3.13}$$

3.6.3 Vektorraum

Das Gesamtgewicht eines einzelnen Token t des Datensatzes $D^{\#}$ berechnet sich nach den vorhergehenden Abschnitten wie folgt:

$$w_{tot}(t, D^{\#}) = \ln\left(1 + \frac{\|\Psi\|}{\|\Psi_t\| + 1}\right) \times \frac{w_f \times tf(D^{\#}, t)}{fsize} \tag{3.14}$$

Wird nun jedem Token aus der Gesamt-Tokenmenge T ein Gewicht in Bezug auf das Dokument D zugeordnet, so erhält man einen Dokumentvektor D^{+}. Damit wird ein Dokument auf in einen allgemeinen Vektorraum abgebildet, der $\|T\|$ Dimensionen hat. Dies stellt die Vektorraumtransformation dar.

$$D^{+} = (D_t^{+})_{t \epsilon T} \, mit \, D_t^{+} = w_{tot}(t, D^{\#}) \tag{3.15}$$

Das Beispiel in Tabelle 3.3 zeigt die Transformation in den Vektorraum für drei Datensätze „PETER“, „PETRA“ und „HANS“. Die Tabelle würde im Normalfall auch die Token aus anderen Feldern als dem Vornamen enthalten, was aber hier aus Platzgründen nicht möglich ist. Die drei Ergebnisvektoren A^{+}, B^{+} und C^{+} ergeben sich jeweils aus den Gesamtgewichten w_{tot} für die Token in den jeweiligen Datensätzen.

Dabei ist anzumerken, dass ein solcher Dokumentvektor nie direkt im System abgebildet wird. Eine solche Darstellung als Vektor wäre für die Berechnung und Speicherung ineffizient. Zum Beispiel lässt sich aus der gegebenen Definition der Kosinus-Ähnlichkeit herauslesen, dass Komponenten mit Gewicht 0 das Ergebnis nicht beeinflussen. In einem Dokument nicht vorkommende Tokens haben generell ein

Beispiel: Vektorraum-Transformation

T		PETER		PETRA		HANS	
t	w_g	w_l	w_{tot}	w_l	w_{tot}	w_l	w_{tot}
ANS	1,42	0,00	0,00	0,00	0,00	0,50	0,71
ETE	1,04	0,33	0,35	0,00	0,00	0,00	0,00
ETR	1,51	0,00	0,00	0,33	0,50	0,00	0,00
HAN	1,07	0,00	0,00	0,00	0,00	0,50	0,53
ÜLL	1,14	0,33	0,38	0,00	0,00	0,00	0,00
LLE	1,22	0,33	0,41	0,33	0,41	0,00	0,00
TRA	1,41	0,00	0,00	0,33	0,47	0,00	0,00
			$\underbrace{\quad}_{A^+}$		$\underbrace{\quad}_{B^+}$		$\underbrace{\quad}_{C^+}$

Tabelle 3.3: Vektorraum-Transformation

Gesamtgewicht von 0 (da $tf(D^{\#}, t) = 0$). Es ist zu erwarten, dass die meisten Token in diese Kategorie fallen, da die Gesamtanzahl der Token in der Regel deutlich größer ist, als die Menge der Token in einem einzelnen Dokument. Es macht daher keinen Sinn, diese Komponenten zu berechnen oder zu speichern.

Auch die Speicherung der Komponenten mit Gewicht größer als 0 würde zu einer unnötigen Speicherbelastung führen. In dem hier vorgestellten System wird daher komplett auf die Speicherung einzelner Komponenten verzichtet. Die Gewichtung wird bei Bedarf jeweils neu berechnet. Dazu benötigte Kerninformationen werden in Indices hinterlegt und bei Bedarf vorberechnet (zum Beispiel die häufig benötigte Dokumentlänge), was eine schnelle Berechnung erlaubt. Diese Indices können zudem die Laufzeit der eigentlichen Dubletten-Erkennung enorm verkürzen.

3.7 Indexerstellung

Da eine paarweise Berechnung der Kosinus-Ähnlichkeit bei größeren Datenbeständen zu rechenaufwendig ist und Blocking Techniken einige Nachteile haben (vgl. Abschnitt 2.4.2), wird in dem hier vorgestellten Algorithmus ein inverser Index benutzt, um von Anfang an den Umfang der zu überprüfenden Datensatzpaare einzuschränken. Dabei

wird der inverse Index nicht nur für die Auswahl von ähnlichen Datensätzen als Blocking-Technik, sondern direkt für die Berechnung der Kosinus-Ähnlichkeit verwendet. Voraussetzung dafür ist zunächst die Erstellung eines Forwärts-Index, der alle wichtigen Informationen über ein Dokument enthält.

3.7.1 Erstellung des Forwärts-Index

Ein Forwärts-Index ordnet jedem Datensatz sein Dokument (also die Liste seiner Token) zu. Zusätzlich wird die Frequenz jedes Tokens im Dokument, die Anzahl der Token in jedem Datensatzfeld und die Länge des dazugehörigen Dokumentvektors gespeichert. Ein Index besteht also aus folgender materialisierter Abbildung:

$$FE(D^{\#}) = \left(\{(t, tf) | t \in D\}, (fsize_f)_{f \in F}, \|D^{+}\|\right) \tag{3.16}$$

Die Berechnung der Länge des Dokumentvektors $\|D^{+}\|$ wird dabei zunächst nicht durchgeführt, da diese Berechnung die idf-Werte aller im Dokument befindlichen Token benötigt. Um den idf-Wert zu ermitteln, müssten zu diesem Zeitpunkt alle anderen Dokumente darauf überprüft werden, ob sie ebenfalls das Token enthalten. Die Indexerstellung hätte damit eine Laufzeit von $O(n^2)$, wenn n die Anzahl der Datensätze insgesamt ist. Deutlich effizienter ist es diese Berechnung erst nach der Erstellung des inversen Index auszuführen. Die Aufteilung des Datensatzes in Token, sowie die Berechnung mit welcher Häufigkeit ein Token in dem Datensatz vorkommt, ist dagegen, genau wie die Berechnung der einzelnen Feldlängen, ausschließlich von der Datensatzlänge abhängig. Die Gesamtlaufzeit ohne Berechnung der Dokumentvektor-Länge beträgt dann $O(ln)$ wobei l die durchschnittliche Datensatzlänge bezeichnet.

3.7.2 Erstellung des inversen Index

Aus dem oben erstellten Forwärts-Index wird nun ein inverser Index erstellt, der einem Token t die Menge der Dokumente zuweist, die dieses Token enthalten. Dabei wird ebenfalls gespeichert wie oft das Token in dem Dokument vorkommt, um die spätere Berechnung der Gewichte im Dokumentvektor zu optimieren.

$$RE(t) = \left(\{(D^{\#}, tf) | D^{\#} \in \Psi \wedge t \in D\}\right) \tag{3.17}$$

$$mit\, D = tokenize(D^{\#}) \tag{3.18}$$

Ein solcher Index-Eintrag wird als Posting-Liste des Tokens t bezeichnet. Die Erstellung des Index kann durch einen einmaligen Durchlauf durch den Forwärts-Index erstellt werden, so dass auch hier die gesamte Laufzeit $O(ln)$ beträgt. Eine wichtige Voraussetzung für den späteren Ablauf ist, dass zwischen allen Datensätzen ein Ordnungskriterium definiert ist und die Reihenfolge der Datensätze in den einzelnen Posting-Listen diesem Ordnungskriterium entspricht. Dies lässt sich ohne zusätzlichen Aufwand realisieren, wenn für jeden Datensatz eine Datensatznummer definiert wird, die der Durchlaufreihenfolge im Forwärts-Index entspricht. Die Laufzeit wird durch diese Maßnahme nicht beeinflusst.

3.7.3 Berechnung der Dokumentlängen

Nach Erstellen des inversen Index kann der Forwärts-Index nun um die Dokumentlänge ergänzt werden.

$$\|D^{+}\| = \sqrt{\sum_{t \in D} D_t^{+2}} \tag{3.19}$$

$$= \sqrt{\sum_{t \in D} w_{tot}(t, D^{\#})^2} \tag{3.20}$$

$$w_{tot}(t, D^{\#}) = idf_t \times \frac{w_f \times tf(D^{\#}, t)}{fsize} \tag{3.21}$$

Zur Berechnung der Länge muss also für jedes Token im Dokument die Gewichtung berechnet werden. Die Token können aus dem Forwärts-Index durchlaufen werden. Die sonstigen für die Berechnung benötigten Komponenten können alle in konstanter Zeit abgerufen werden, wenn davon ausgegangen wird, dass der Abruf eines einzelnen Index-Eintrags konstant ist und die Größe aller Listen und Mengen ebenfalls in konstanter Zeit ermittelt werden kann:

idf_t benötigt zur Berechnung die Gesamtanzahl der Datensätze sowie die Datensätze die das Token t enthalten. Ersteres kann über

die Länge des Forwärts-Index ermittelt werden. Die Anzahl der Dokumente die das Token t enthalten, kann dagegen über den inversen Index ermittelt werden, wenn zunächst der entsprechende Eintrag aus dem Index geladen wird und dann die Länge der dort gespeicherten Posting-Liste verwendet wird.

w_f ist ein konstanter Wert für das Feld f und benötigt keine weitere Berechnungsschritte.

$tf(D^{\#}, t)$ wird mit den Token zusammen im Forwärts-Index gespeichert und ist daher beim Durchlaufen dieser Liste unmittelbar verfügbar.

$fsize$ ist ebenfalls als Teil des Forwärts-Index hinterlegt. Auch hier wird davon ausgegangen, dass der zu dem Feld gehörende Eintrag in konstanter Zeit abgerufen werden kann.

Die Gesamtzeit für diesen Berechnungsschritt ist also ebenfalls $O(ln)$ (l sei wieder die durchschnittliche Anzahl an Token pro Dokument und n die Anzahl der Dokumente).

3.8 Iterative Dublettenerkennung

Nach der Erstellung des inversen Index beginnt die eigentliche Suche nach Dubletten. Dabei wird jeder Datensatz getrennt betrachtet und eine Liste aller potenziellen Dubletten erstellt. Die Liste potenzieller Kandidaten wird durch die Suche im inversen Index bestimmt. Dazu wird zunächst die Liste aller Token aus dem Forwärts-Index geladen, die in dem zu untersuchenden Quelldatensatz vorkommen.

Für jedes Token t wird nun nach und nach die Posting-Liste aus dem inversen Index geladen, die alle Datensätze $B^{\#}$ auflistet, die ebenfalls das Token enthalten, sowie die Anzahl der Vorkommen von t in $B^{\#}$ ($tf(B^{\#}, t)$). Ein spezielles Pruning Verfahren (später beschrieben) sorgt dafür, dass nur die notwendige Anzahl an Index-Einträgen geladen wird, mit der noch sichergestellt werden kann, dass keine potenziellen Kandidaten übersprungen werden.

Die einzelnen Posting-Listen werden zusammengeführt und sortiert, so dass gleiche Datensätze aus unterschiedlichen Listen zusammen stehen. Da die einzelnen Listen im inversen Index bereits vorsortiert sind, lässt sich dies sehr effizient durchführen.

Um die spätere Berechnung weiter zu optimieren, wird zu jedem Eintrag zusätzlich ein Schlüssel gespeichert, der neben dem Token zu dem der Eintrag gehört, auch das globale Gewicht und das Gesamtgewicht für das Token im Quelldatensatz enthält.

Ein einzelner Eintrag hat dann folgende Struktur:

$$(\overbrace{(t, w_g, w_{totA})}^{\text{Schlüssel}}, \overbrace{(B^{\#}, tf(t, B^{\#}))}^{\text{Wert}}) \tag{3.22}$$

Die resultierende Liste enthält damit alle Datensätze, die mindestens ein gemeinsames Token mit dem Quelldatensatz besitzen und alle für die Berechnung der Ähnlichkeit notwendigen Daten.

Token, die nur in einem der beiden zu vergleichenden Dokumente vorkommen, müssen bei der Berechnung nicht berücksichtigt werden, weil sie keinen Einfluss auf das Ergebnis haben. Dies ergibt sich aus der Formel der Kosinus-Ähnlichkeit: Das Gesamtgewicht w_{tot} für ein Token in einem Dokument nimmt den Wert 0, wenn das Token in dem Dokument nicht vorkommt. Das Produkt der Gewichte aus beiden Dokumente im Zähler ist dann ebenfalls 0 und beeinflusst die Summe nicht. [4]

Die resultierende Liste kann nun durchlaufen und die Übereinstimmung mit jedem Dokument berechnet werden. Dies führt zu dem folgenden vereinfachten Algorithmus, der die Berechnung der Ähnlichkeit für

[4] Die Gewichte haben natürlich einen Einfluss auf die Dokumentvektorlänge $\|D^+\|$, die aber bereits im Vorhinein berechnet wurde und aus dem Vorwärts-Index ausgelesen werden kann.

die einzelnen Dokumente durchführt, aber noch nicht mittels Pruning optimiert wurde:

Eingabe: $A^{\#}$, Zusammengeführte Liste *queue*, Schwellwert σ
Ausgabe: Liste von Tupeln $(B_i^{\#}, sim_i)$ mit $sim_i > \sigma$
$((t, w_g, w_{totA}), (B^{\#}, tf(t, B^{\#}))) = queue.pop();$
while *queue is not empty* **do**
  $B^{\#\prime} = B^{\#};$
  repeat
    $w_{totB} = w_l(B^{\#}, t, tf(t, B^{\#})) * w_g;$
    $sumAB = sumAB + (w_{totB} \times w_{totA});$
    $((t, w_g, w_{totA}), (B^{\#}, tf(t, B^{\#}))) = queue.pop();$
  until $B^{\#} \neq B^{\#\prime}$;
  $sim = \dfrac{sumAB}{\|\vec{B^+}\| \times \|\vec{B^+}\|};$
  if $sim >= \sigma$ **then**
    yield $(B^{\#\prime}, sim);$
  end
end

3.8.1 Pruning

Ein wichtiger Schritt zur Optimierung der Index-Abfrage ist das Pruning, also das vorzeitige Ausschließen von Kandidaten die, mathematisch beweisbar, unmöglich eine höhere Ähnlichkeit mit dem zu untersuchenden Datensatz aufweisen können, als durch einen vom Benutzer vorgegebenen Schwellwert festgelegt wurde.

Das Pruning setzt bereits bei dem Laden der Posting-Listen für einen bestimmten Datensatz an. Um das Verfahren zu verstehen, kann zunächst ein vereinfachtes Modell betrachtete werden, bei dem das Gesamtgewicht w_{tot} eines Token in einem Datensatz den Wert 0 annimmt, wenn das Token nicht in dem Datensatz vorkommt oder 1, wenn das Token vorkommt. Es soll außerdem angenommen werden, dass ein Token in jedem Datensatz nur einmal vorkommen kann und das die Gesamtanzahl der Token für alle Datensätze gleich ist (l).

Die Kosinus-Ähnlichkeit für zwei Datensätze $A^{\#}$ und $B^{\#}$ mit den dazugehörigen Tokenlisten A und B ist dann definiert als:

$$\cos(\sigma) = \frac{\|\{t|t \in A \wedge t \in B\}\|}{l} \tag{3.23}$$

Die Anzahl der Token, die in beiden Datensätzen vorkommen, wird also geteilt durch die gemeinsame Länge l. Das entspricht dem prozentualen Anteil der Token aus Datensatz $A^{\#}$, die auch im Datensatz $B^{\#}$ vorkommen.

Nun legen wir einen Schwellwert σ fest, der angibt, ab wie viel Prozent Übereinstimmung es sich bei den beiden Datensätzen möglicherweise um Dubletten handelt.

$$\frac{\|\{t|t \in A \wedge t \in B\}\|}{l} \geq \sigma \tag{3.24}$$

Daraus folgt:

$$\|\{t|t \in A \wedge t \in B\}\| \geq \sigma \times l \tag{3.25}$$

Für zwei Datensätze mit Länge 10 und einem Schwellwert σ von 80% müssten also mindestens 8 Token übereinstimmen, damit diese als Dublette markiert werden. Im Umkehrschluss können alle Datensätze ausgeschlossen werden, die in mehr als $10 - 8 = 2$ Termen keine Übereinstimmung mit dem Zieldatensatz haben.

Betrachten wir folgenden Dokument der Länge 10:

$$A^{\#} = \{DatensatzABC\} \tag{3.26}$$

$$A = \{DAT, ATE, TEN, ENS, NSA, \tag{3.27}$$

$$SAT, ATZ, TZA, ZAB, ABC\} \tag{3.28}$$

$$l = \|A\| = 10 \tag{3.29}$$

$$\sigma = 80\% \tag{3.30}$$

Ein Datensatz $B^{\#}$ mit identischer Länge l kann den Schwellwert σ nur dann erreichen, wenn seine Tokenliste mindestens eines der drei ersten Token DAT, ATE bzw. TEN enthält. Werden ähnliche Datensätze mit Hilfe eines inversen Index ermittelt, so müssen zunächst nur

die Posting Listen für diese Token geladen werden. Ähnliche Datensätze müssen sich innerhalb dieser Listen befinden.

Durch das Laden weiterer Posting Listen für die folgenden Tokens können nun weitere Dokumente ausgeschlossen werden. Wurden die ersten vier Posting Listen geladen, so können alle Dokumente ausgeschlossen werden, die lediglich einmal in diesen Listen auftauchen, weil diese in mindestens drei der Token nicht übereinstimmen. Die Suche kann abgeschlossen werden, sobald alle Dokumente auf diese Weise ausgeschlossen wurden (dann wurde keine Dubletten gefunden) oder alle Posting-Listen geladen wurden und es mindestens ein Dokument gibt, bei dem die Ähnlichkeit 80% oder mehr beträgt (also eine Dublette gefunden wurde).

Das Verfahren kann auf die Kosinus-Ähnlichkeit mit tf-idf Gewichtung und mit unterschiedlichen Längen ausgeweitet werden. Die Ähnlichkeit zwischen dem Datensatz $A^{\#}$ für das die Posting-Listen geladen werden und einem anderen Datensatz $B^{\#}$ beträgt allgemein:

$$sim = \frac{A^{+} \cdot B^{+}}{\|A^{+}\| \|B^{+}\|} \tag{3.31}$$

Befindet sich in den ersten k geladenen Posting Listen von $A^{\#}$ kein Eintrag für $B^{\#}$, so bedeutet das, dass die dazugehörigen Komponenten im Dokumentenvektor B^{+} gleich 0 sind:

$$A^{+}_{<} = (A^{+}_{0}, A^{+}_{1}, ...A^{+}_{k-1}) \qquad A^{+}_{>} = (A^{+}_{k}, A^{+}_{k+1}, ...A^{+}_{n}) \tag{3.32}$$

$$B^{+}_{<} = (B^{+}_{0}, B^{+}_{1}, ...B^{+}_{k-1}) \qquad B^{+}_{>} = (B^{+}_{k}, B^{+}_{k+1}, ...B^{+}_{n}) \tag{3.33}$$

$$B^{+}_{<} = \vec{0} \tag{3.34}$$

$$\tag{3.35}$$

Dies resultiert in einer oberen Schranke $max(sim)$ für die beiden Vektoren A^+ und B^+:

$$A^+ \cdot B^+ = A^+_> \cdot B^+_> \tag{3.36}$$

$$\|B^+\| = \|B^+_>\| \tag{3.37}$$

$$sim = \frac{A^+_> \cdot B^+_>}{\|A^+\|\|B^+_>\|} \tag{3.38}$$

$$= \frac{A^+_> \cdot B^+_>}{\|A^+_>\|\|B^+_>\|} \times \frac{\|A^+\|}{\|A^+_>\|} \tag{3.39}$$

$$max(sim) = \underbrace{max(\frac{A^+_> \cdot B^+_>}{\|A^+_>\|\|B^+_>\|})}_{=1 \text{ für } A^+_>=B^+_>} \times \frac{\|A^+\|}{\|A^+_>\|} \tag{3.40}$$

$$max(sim) = \frac{\|A^+\|}{\|A^+_>\|} \tag{3.41}$$

$$= \frac{\sqrt{\sum_{i=k}^{n} A_i^{+2}}}{\|A^+\|} \tag{3.42}$$

$$= \boxed{\frac{\sqrt{\|A^+\|^2 - \sum_{i=0}^{k} A_i^{+2}}}{\|A^+\|}} \tag{3.43}$$

Es müssen also so viele Listen geladen werden, dass $max(sim) < \sigma$. Das lässt sich iterativ leicht lösen:

k=0;
unloadedLength = $\|A^+\|^2$;
repeat
  maxSim = $\frac{\sqrt{unloadedLength}}{\|A^+\|}$;
  unloadedLength = $unloadedLength - w_{tot}(A_k, A^\#)^2$;
  k=k+1;
  queue.push((t, w_g, w_{tot}), $RE(A^+_k)$);
until $maxSim < \sigma$;

Für jeden Datensatz, der in den ersten k Listen gefunden wurde, müssen nun weitere Posting-Listen geladen werden, solange bis entweder alle Posting-Listen geladen wurden, oder auf Grund der geladenen Posting-Listen einwandfrei hervorgeht, dass der Datensatz keine Dublet-

te des Originaldatensatzes sein kann. Dabei gehen wir erneut iterativ vor:

Eingabe: $A^{\#}$, Zusammengeführte Liste *queue*, Schwellwert σ
Ausgabe: Liste von Tupeln $(B_i^{\#}, sim_i)$ mit $sim_i > \sigma$
k=0;
unloadedLength = $\|A^+\|^2$;
repeat
  maxSim = $\frac{\sqrt{unloadedLength}}{\|A^+\|}$;
  unloadedLength = $unloadedLength - w_{tot}(A_k, A^{\#})^2$;
  k=k+1;
  queue.push((t, w_g, w_{tot}), $RE(A_k^+)$);
until $maxSim < \sigma$;
$((t, w_g, w_{totA}), (B^{\#}, tf(t, B^{\#}))) = queue.pop()$;
while *queue is not empty* **do**
  $B^{\#\prime} = B^{\#}$;
  $bLengthRemaining = \|B^+\|^2$;
  repeat
    $w_{totB} = w_l(B^{\#}, t, tf(t, B^{\#})) * w_g$;
    $sumAB = sumAB + (w_{totB} \times w_{totA})$;
    $bLengthRemaining = bLengthRemaining - w_{totB}^2$;
    `/* Berechnung der „maximalen Ähnlichkeit" */`
    maxSim = $\frac{sumAB + \sqrt{unloadedLength \times bLengthRemaining}}{\|A^+\| \times \|B^+\|}$;
    `/* Wenn nötig weitere Posting Listen laden */`
    while $maxSim \geq \sigma$ **do**
      unloadedLength = $unloadedLength - w_{tot}(A_k, A^{\#})^2$;
      k=k+1;
      queue.push((t, w_g, w_{tot}), $RE(A_k^+)$);
      maxSim = $\frac{sumAB + \sqrt{unloadedLength \times bLengthRemaining}}{\|A^+\| \times \|B^+\|}$;
    end
    $((t, w_g, w_{totA}), (B^{\#}, tf(t, B^{\#}))) = queue.pop()$;
  until $B^{\#} \neq B^{\#\prime}$;
  $sim = \frac{sumAB}{\|\vec{B^+}\| \times \|\vec{B^+}\|}$;
  if $sim >= \sigma$ **then**
    yield $(B^{\#\prime}, sim)$;
  end
end

In diesem Algorithmus wurde nun das Laden der Posting-Listen integriert. Wichtigste Änderung ist dabei das Laden weiterer Posting-Listen, wenn nicht ausgeschlossen werden kann, dass ein Datensatz durch diese zusätzlichen Posting-Listen die erforderliche Ähnlichkeit mit dem Quelldatensatz erreichen kann.

Dazu wird nach jedem Berechnungsschritt für einen Datensatz $B^{\#}$ erneut der Wert $max(sim)$ ermittelt, der angibt, welche maximale Ähnlichkeit zu dem Quelldatensatz $A^{\#}$ dieser erreichen kann, unter der Annahme, dass unter den geladenen k Posting-Listen keine weiteren Übereinstimmungen mehr zu finden sind. Diese Annahme macht Sinn. Denn wenn noch Übereinstimmungen in den geladenen Posting-Listen existieren, werden diese in den nächsten Durchläufen berücksichtigt und der $max(sim)$-Wert neu berechnet. In diesem Fall liegt der neue Wert auf jeden Fall höher, so dass mindestens die gleiche Anzahl oder mehr Posting-Listen geladen werden müssten. Die Annahme beeinflusst daher nicht das korrekte Ergebnis und erleichtert die Programmlogik.

Die Korrektheit des $maxSim$-Werts als obere Schranke für die Ähnlichkeit zwischen $A^{\#}$ und $B^{\#}$ lässt sich wie folgt beweisen. Wir teilen die Vektoren erneut an dem Index k in zwei separate Vektoren auf:

$$A^+_< = (A^+_0, A^+_1, ...A^+_{k-1}) \qquad A^+_> = (A^+_k, A^+_{k+1}, ...A^+_n) \tag{3.44}$$

$$B^+_< = (B^+_0, B^+_1, ...B^+_{k-1}) \qquad B^+_> = (B^+_k, B^+_{k+1}, ...B^+_n) \tag{3.45}$$

$$B^+_< = \vec{0} \tag{3.46}$$

$$\tag{3.47}$$

Die Kosinus-Ähnlichkeit sim zwischen den Vektoren A^+ und B^+ ist dann:

$$sim = \frac{A^+_< \cdot B^+_< + A^+_> \cdot B^+_>}{\|A^+\|\|B^+\|} \tag{3.48}$$

Bis auf $A^+_> \cdot B^+_>$ sind dabei alle Werte bekannt[5]. Es gilt daher:

$$max(sim) = \frac{A^+_< \cdot B^+_< + max(A^+_> \cdot B^+_>)}{\|A^+\|\|B^+\|} \tag{3.49}$$

$$max\Big(\frac{A^+_> \cdot B^+_>}{\|A^+_>\|\|B^+_>\|}\Big) = 1 \tag{3.50}$$

$$max(A^+_> \cdot B^+_>) = \|A^+_>\|\|B^+_>\| \tag{3.51}$$

$$max(sim) = \boxed{\frac{A^+_< \cdot B^+_< + \|A^+_>\|\|B^+_>\|}{\|A^+\|\|B^+\|}} \tag{3.52}$$

$$\tag{3.53}$$

3.8.2 Laufzeit

Die Laufzeit des Algorithmus ist von der durchschnittlichen Anzahl von Token in einem Dokument l, sowie der durchschnittlichen Länge der Posting-Listen p abhängig und beträgt somit $O(lp)$. Da dieser Algorithmus für jedes Dokument einmal aufgerufen wird, beträgt die asymptotische Gesamtlaufzeit $O(lnp)$. Die Vorverarbeitungsschritte liegen ebenfalls innerhalb dieser Komplexitätsklasse. Das Pruning Verfahren bietet einen entscheidenden, aber konstanten Faktor, um die reale Laufzeit zu reduzieren. Genaue Zeitmessungen werden im Abschnitt 5.3 vorgestellt.

3.9 Nachverarbeitung und Gruppierung der Ergebnisse

Der vorangegangene Algorithmus stellt eine vereinfachte Form des tatsächlich implementierten Algorithmus dar. Er wird nacheinander für alle Datensätze im System aufgerufen, um die möglichen Dubletten für einen Datensatz zu ermitteln. Das Ergebnis ist dann eine Menge von Tupeln der Form $(A^\#, B^\#, sim)$, das die beteiligten Datensätze und den ermittelten Ähnlichkeitswert sim enthält.

Da die Ähnlichkeits-Relation symmetrisch ist, gibt es für jedes Paar von Datensätzen, die als Dubletten erkannt wurden, immer zwei

[5] $A^+_< \cdot B^+_<$ entspricht der aktuellen Summe $sumAB$ im Algorithmus, wenn – wie ja angenommen wurde – keine weiteren Übereinstimmungen in den ersten k Komponenten der Vektoren existieren. Die Längen $\|A^+\|$ $\|B^+\|$ sind im Index abrufbar.

Ergebnisse $(A^{\#}, B^{\#}, sim)$ und $(B^{\#}, A^{\#}, sim)$. Diese redundanten Paare können leicht herausgefiltert werden. Das lässt sich schon im Algorithmus realisieren, indem alle Datensätze in den Posting-Listen ignoriert werden, die bereits vorher als Quelldatensatz überprüft wurden. Als weiterer positiver Effekt wird durch das Überspringen dieser Datensätze die Laufzeit verringert.

Ein weiterer Schritt zur Aufbereitung der Ergebnisse stellt die Gruppierung der gefundenen Ergebnispaare dar. Eine Gruppe von n Datensätzen, die allesamt Dubletten der jeweils anderen Datensätze sind, resultiert in $n * (n-1)$ Ergebnispaaren. Eine Gruppe aus vier Dubletten verursacht also 12 Ergebnispaare. Um die Ergebnisse übersichtlicher zu präsentieren, können diese Ergebnispaare zu einer einfachen Gruppe aus 4 Elementen zusammengefasst werden. Die Ähnlichkeit wird dazu, der Einfachheit halber, als Äquivalenzrelation aufgefasst, so dass $A^{\#} \sim B^{\#}$ und $B^{\#} \sim C^{\#}$ auch automatisch $A^{\#} \sim C^{\#}$ impliziert.

Dies ist mathematisch nicht ganz korrekt, da die Ähnlichkeit zwischen $A^{\#}$ und $C^{\#}$ unterhalb des Schwellwertes σ liegen kann und es sich damit nach unserer Definition nicht um Dubletten handeln würde. Es vereinfacht aber die Darstellung für den Benutzer und führt in der Praxis selten zu Problemen. Die Gruppierung lässt sich leicht über passende Algorithmen der Graphentheorie erstellen und wird hier nicht näher beschrieben.

4 Implementierung

Während im letzten Kapitel das theoretische Fundament der Dublettenerkennung im Vordergrund stand, soll nun die konkrete Umsetzung der Konzepte beschrieben werden. Auf Grund der umfangreichen Implementierung kann hier nur ein grober Überblick gegeben werden. Für die Dublettenerkennung irrelevanten Bestandteile (z.B. zur Benutzerauthentifizierung) wurden daher bewusst ausgelassen.

4.1 Übersicht

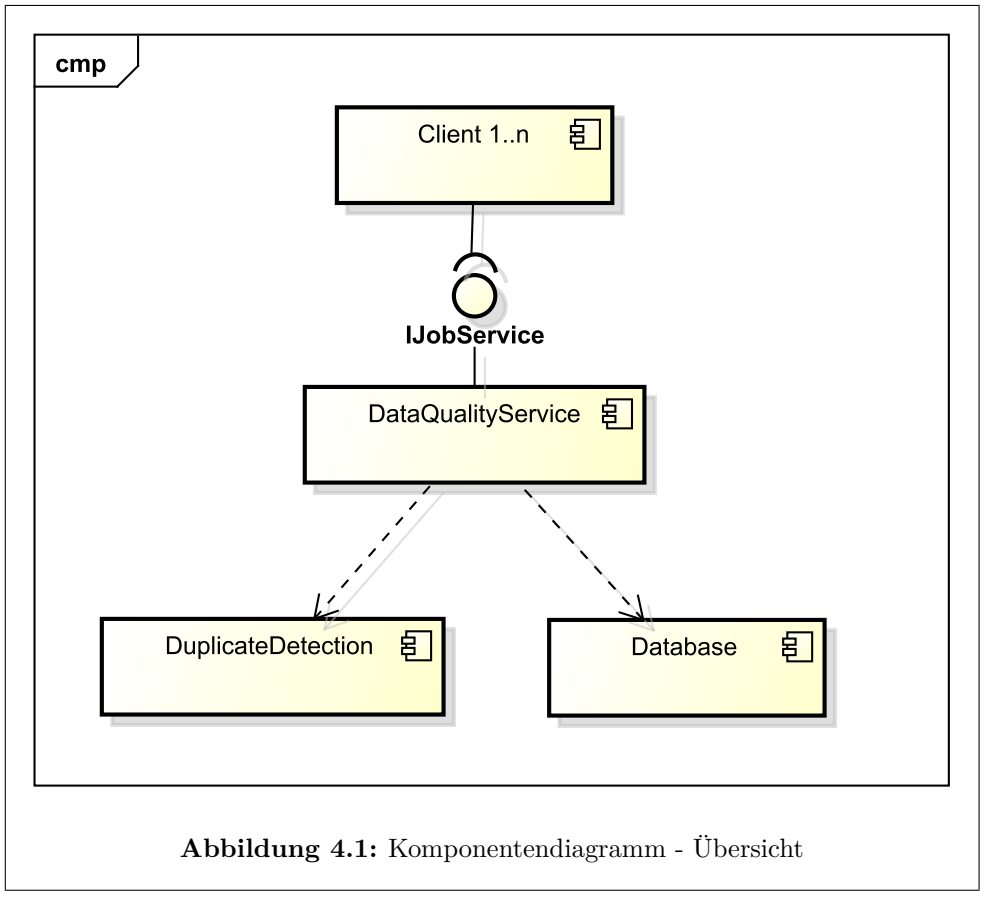


Abbildung 4.1: Komponentendiagramm - Übersicht

Die Implementierung besteht aus zwei Komponenten. Neben einer allgemeinen Bibliothek für die Dublettenerkennung, wurde ein Web-

dienst entworfen, der Datensätze und Aufträge zur Dublettensuche entgegennimmt und die genannte Bibliothek nutzt, um diese auszuführen. Beide Komponenten sind in C# entwickelt worden und benötigt zur Ausführung das .NET Framework in der Version 4.0.

Da der Webdienst das standardisierte SOAP-Protokoll benutzt, können beliebige Clients auf den Webdienst zugreifen und die Dublettenerkennung über einen zentralen Server ausführen, solange sie SOAP als Protokoll unterstützen. Dies ist für alle gängigen Entwicklungsplattformen der Fall. Alternative Protokolle wie JSON können mit wenigen Änderungen ebenfalls benutzt werden. Es ist daher nicht nötig, dass die Clients ebenfalls in .NET geschrieben sind.

Zur Speicherung der Eingabedaten und Ergebnisse wird ein Objektrelationale Mapper von Microsoft (Entity Framework) verwendet, der bereits im .NET Framework integriert ist. Die Daten werden in einer SQL Server Datenbank gespeichert, wobei prinzipiell alle Datenbanksysteme mit Entity Framework-Unterstützung verwendet werden können.

4.2 Bibliothek zur Dublettenerkennung

Die Bibliothek zur Dublettenerkennung implementiert den im Kapitel „**Entwurf**“ vorgestellten Algorithmus.

Zum Ausführen der Dublettenerkennung wird die Klasse `DuplicateDetector` verwendet. Listing 4.1 zeigt die Verwendung der Klasse anhand eines Unit-Tests.

```
1  public class Address
2  {
3      public Address(int id, string name, string
           street, string postalCode, string city)
4      {
5          Id = id;
6          Name = name;
7          Street = street;
8          PostalCode = postalCode;
9          City = city;
10     }
11
12     public int Id { get; set; }
13     public string Name { get; set; }
14     public string Street { get; set; }
15     public string PostalCode { get; set; }
```

```
16      public string City { get; set; }
17  }
18
19  [TestMethod]
20  public void SimpleDuplicateDetectorTest()
21  {
22      //Erstellen des Dubletten-Detektors
23      // Ähnlichkeitsschwelle: 80%
24      // Vollständigkeit: 100%
25      // => Adressen mit NULL-Werten ignorieren
26      var dd = new DuplicateDetector<Address,string ↙
            >(0.8, 1);
27
28      //Anlegen der Testadressen, Adressen 1 und 3  ↙
            sind Dubletten
29      var address1 = new Address(1, "Peter Müller", " ↙
            Hauptstr. 123", "51274", "Murkshausen");
30      var address2 = new Address(2, "Hans Dampf", " ↙
            Feldweg. 2", "43921", "Pappstadt");
31      var address3 = new Address(3, "Müller, Peter",  ↙
            "Hauptstraße 123a", "51272", "Murkshausen");
32
33      //Zerlegungsfunktion für Trigramme
34      var tok = new StringToTrigramTokenizer();
35
36      //Felder und Gewichtung definieren
37      dd.AddFieldDefinition(a => a.Name, tok, 1);
38      dd.AddFieldDefinition(a => a.Street, tok, 1);
39      dd.AddFieldDefinition(a => a.PostalCode, tok,  ↙
            0.3);
40      dd.AddFieldDefinition(a => a.City, tok, 0.7);
41
42      //Zu überprüfende Adressen hinzufügen
43      dd.Add(new[] {address1, address2, address3});
44
45      //Erkennung ausführen
46      var result = dd.FindDuplicates().ToArray();
47
48      //Erwartet: Adresse 1 und 3 werden als Dublette ↙
            erkannt
49      Assert.AreEqual(1, result.Length);
50      Assert.AreEqual(address1, result[0].Item1);
51      Assert.AreEqual(address3, result[0].Item2);
52  }
```

Listing 4.1: Komponententest der Klasse DuplicateDetector

Die Klasse `Address` repräsentiert die Entitäten für die wir eine Dublettenerkennung durchführen wollen. `DuplicateDetector` unterstützt

beliebige Klassen, ohne dass diese eine bestimmte Schnittstelle implementieren müssen.

Für die Initialisierung von `DuplicateDetector` werden zwei Typ-Parameter benötigt (vgl. Zeile 26). Als erster Typ-Parameter wird die Entitäten-Klasse angegeben, damit alle Methodenaufrufe und Rückgabewerte auf diese Klasse spezialisiert werden können. Der zweite Parameter gibt den Typ der verwendeten Token an. Dies macht in Ausnahmefällen Sinn, wenn statt der üblichen Trigramme andere Tokens verwendet werden sollen. In den meisten Fällen sollte der Typ `string` verwendet werden.

Als Konstruktorparameter erwartet die Klasse die Schwellwerte für minimale Ähnlichkeit und die erforderliche Vollständigkeit der Entitäten (vgl. σ und ν im Kapitel „**Entwurf**“). In diesem Fall wird also eine Ähnlichkeit von mindestens 80% verlangt, um zwei Datensätze als Dublette zu markieren. Die Entitäten dürfen außerdem keine *NULL*-Werte enthalten, da eine Vollständigkeit von 100% verlangt wird.

Im nächsten Schritt (Zeile 29-31) werden nun einige Beispieladressen angelegt. Es ist leicht zu erkennen, dass es sich bei der letzten Adresse höchstwahrscheinlich um eine Dublette der ersten Adresse handelt. In dem Test soll überprüft werden, ob dies von der Bibliothek auch erkannt wird.

Wie eingangs erwähnt, gibt es keine besonderen Voraussetzungen für Entitäten, damit diese für die Dublettenerkennung verwendet werden können. Da die Dublettenerkennung jedoch die einzelnen Felder der Entität zur Ausführung benötigt, müssen diese der Klasse bekannt gemacht werden. Dazu werden anonyme Funktionen verwendet, die den Feldinhalt extrahieren können. Über die in C# verfügbaren Lambda Ausdrücke lassen sich solche Funktionen in sehr kompakter und lesbarer Form darstellen. Der Ausdruck `a => a.Name` in Zeile 37 steht für eine Funktion, die aus einer als Parameter `a` übergebenen Entität vom Typ `Address` die Eigenschaft `Name` extrahiert. Statt einfacher Eigenschaften können natürlich auch beliebig komplexe Ausdrücke oder Aufrufe zu anderen Funktionen verwendet werden.

Die Funktion `AddFieldDefinition` enthält einen Typparameter der den Wertetyp des Feldes angibt. Dieser wird in den meisten Fällen automatisch durch den Compiler aus dem Rückgabewert der Extrahierungsfunktion ermittelt und muss daher nicht spezifiziert werden.

```
1 public void AddFieldDefinition<TField>(
2         Converter<TEntity,TField> fieldExtractor,
3         ITokenizer<TField,TToken> tokenizer,
4         double weight
5 )
```

Listing 4.2: DuplicateDetector.AddFieldDefiniton - Methode

Nach dem Ermitteln eines Feldwertes, muss dieser in einzelne Token zerlegt werden. Als zweiter Parameter von `AddFieldDefinition` wird dazu ein Objekt übergeben, das die Schnittstelle `ITokenizer` implementiert. Damit kann für jedes Feld gesondert festgelegt werden, wie die Termzerlegung zu erfolgen hat. Auch `ITokenizer` ist generisch entworfen und enthält Typparameter für die Eingabe (den Feldtyp) und die Ausgabe (der Typ der erstellten Token).

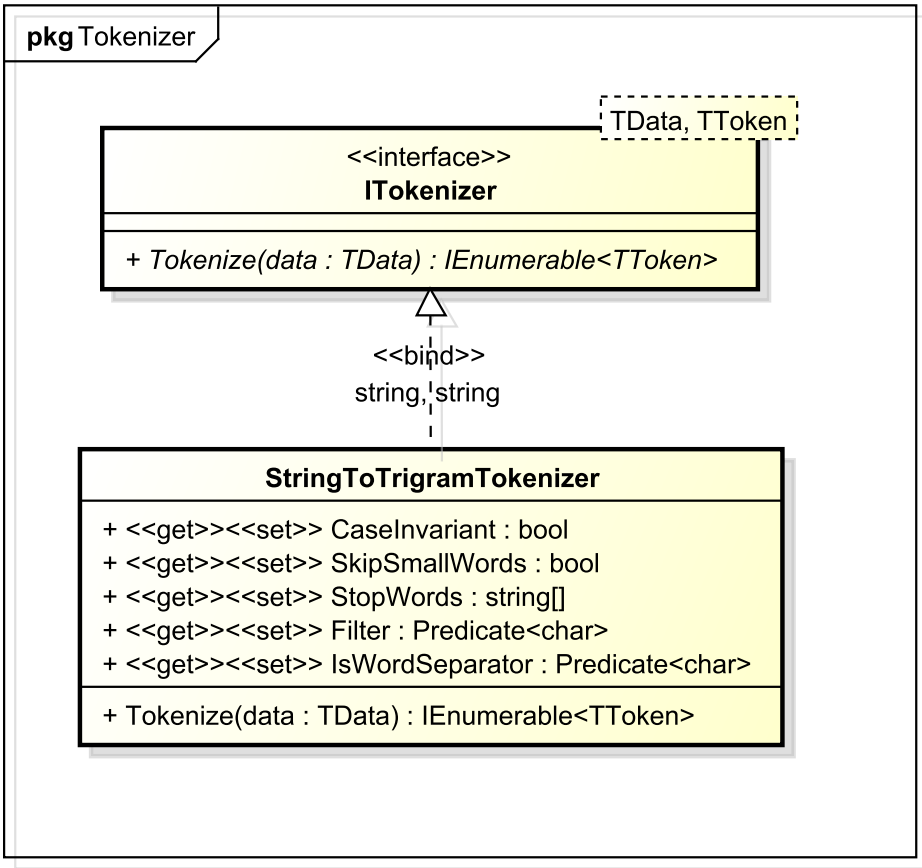


Abbildung 4.2: Komponentendiagramm - DataQualityService

Der in der Bibliothek enthaltene `StringToTrigramTokenizer` implementiert die im Entwurf beschriebene Variante, Zeichenketten in eine Liste von Trigrammen umzuwandeln (vgl. Zeile 34 in Listing 4.1).

Neben der Termzerlegung, wie sie in Abschnitt 3.5 beschrieben wurde, übernimmt diese Klasse auch bestimmte Normalisierungen, wie das Entfernen von Satzzeichen oder die Vereinheitlichung von Groß- und Kleinschreibung, wenn das gewünscht wird. Statt also eine explizite Normalisierung (wie in Abschnitt 3.3) durchzuführen und anschließend die Tokens aus der normalisierten Zeichenketten zu erstellen, erfolgt dies implizit während der Zerlegung, in dem bestimmte Zeichen übersprungen oder umgewandelt werden. Dies verhindert das Erstellen von vielen, nur kurz genutzten, Zeichenketten im Hauptspeicher und reduziert damit den Speicherbedarf. Spezielle Normalisierungen, wie das Zusammenfassen von verschiedenen Feldern, können durch eine angepasste Extrahierungsfunktion (s. O.) umgesetzt werden.

Als letzter Parameter der Funktion `AddFieldDefinition` wird das Feldgewicht (w_f) angegeben. Entscheidend sind dabei weniger die absoluten Werte, sondern die Relation zu den anderen Gewichten. In dem angegebenen Beispiel haben also Postleitzahl und Ort zusammen etwa das gleiche Gewicht wie der komplette Name oder die Straße.

Sobald alle Felder definiert wurden, können die eigentlichen Entitäten über die Methode `Add` hinzugefügt werden. Es ist wichtig, dass alle benötigten Felder vorher definiert werden, da bereits beim Hinzufügen der Entitäten der Forwärts-Index erzeugt wird, der die ermittelten Token für jede Entität speichert.

Die `Add`-Methode erwartet ein `IEnumerable<T>` vom Entitätstyp `T`, also eine Sequenz von Entitäten, die auf Dubletten überprüft werden soll. Die Methode kann bei Bedarf mehrmals ausgeführt werden, um weitere Entitäten hinzuzufügen. In diesem Schritt wird bereits die Tokenzerlegung durchgeführt und die Entitäten im Forwärts-Index abgelegt (vgl. Unterabschnitt 3.7.1).

Sobald alle Entitäten erfasst wurden, kann die eigentliche Dublettenerkennung über die Methode FindDuplicates erfolgen. Dabei wird zunächst der inverse Index erstellt und dann die Dublettenerkennung in mehreren parallelen Threads ausgeführt. Die Methode gibt einen Enumerator (`IEnumerable<>`) zurück, der die gefundenen Dubletten durchläuft. Mit Hilfe des Enumerators können Dubletten bereits verarbeitet werden,

auch wenn die Dublettenerkennung noch nicht vollständig abgeschlossen ist. Der Enumerator blockiert dann gegebenenfalls den aufrufenden Thread solange bis eine weitere Dublette gefunden wurde oder die Suche abgeschlossen ist.

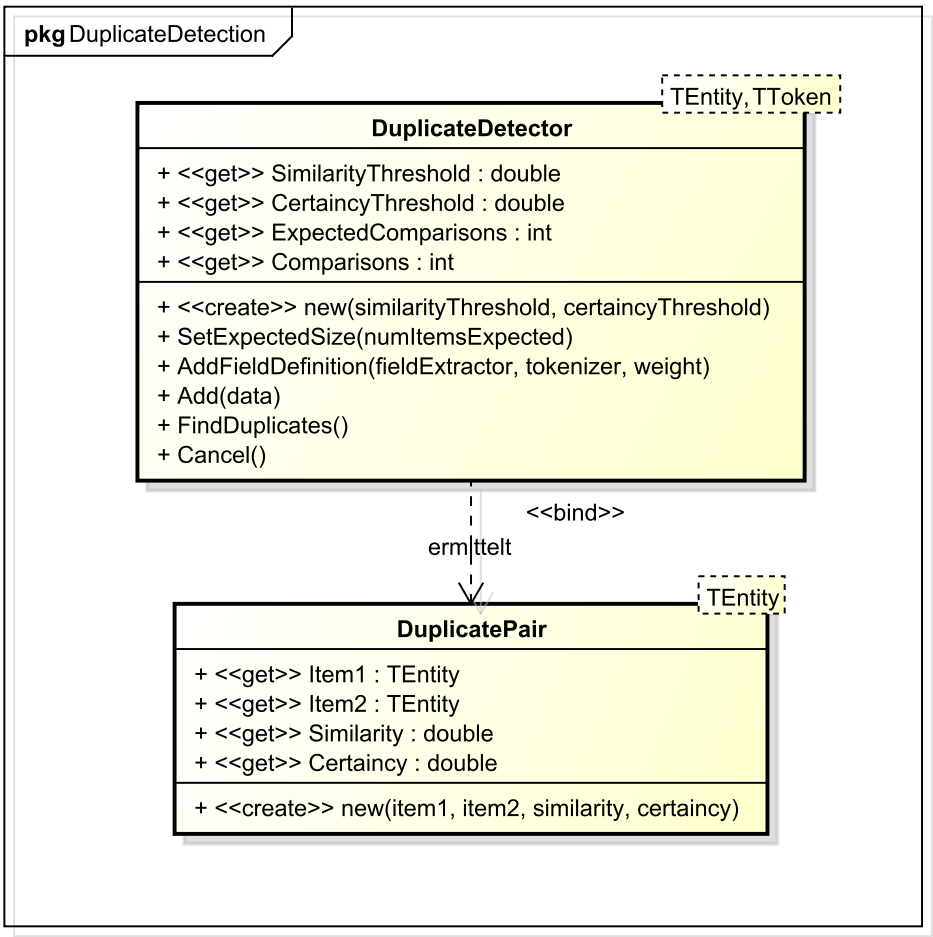


Abbildung 4.3: Klassendiagramm - DuplicateDetector und DuplicatePair

Die zurückgegebenen Dubletten werden durch die Klasse `DuplicatePair` repräsentiert. Diese speichert das Paar der Entitäten sowie deren Grad der Übereinstimmung (`Similarity`). Die Eigenschaft `Certaincy` gibt an welche Aussagekraft die Angabe zur Ähnlichkeit hat. Dies wird aus der Vollständigkeit der beiden beteiligten Datensätze ermittelt.

4.2.1 Indizierung

Die oben angesprochene Indizierung erfolgt nicht direkt in der Klasse `DuplicateDetector`. Stattdessen verwendet `DuplicateDetector` intern die

Klasse `FieldBasedIndex`, die feldbasierte Entitäten indiziert. Diese ist wiederum von der abstrakten Basis-Klasse `Index` abgeleitet, die grundlegende Funktionen zur Indizierung und zum Auffinden von Entitäten bereitstellt.

Die Implementierung der einzelnen Methoden entspricht dem im Entwurf vorgestelltem Algorithmus und wird daher nicht im Detail erläutert. Kernstück bilden die Methoden `Add` zum Hinzufügen neuer Dokumente zum Forwärts-Index, `Build` zum Erstellen des inversen Index, sowie `FindMatches`, die den in Abschnitt 3.8 beschriebenen Algorithmus umsetzt.

4.2.2 Gruppierung

Oft ist es wünschenswert die gefundenen Dublettenpaare zu gruppieren (vgl. Abschnitt 3.9, um Dubletten aus mehr als zwei Datensätzen zusammen zu betrachten (bei drei Datensätzen gibt es sonst bis zu drei verschiedene Dublettenpaare, bei vier Datensätzen schon sechs verschiedene Paare). In der implementierten Bibliothek lässt sich dies über die Klasse `DuplicateGrouping` erreichen. Diese nimmt die Ergebnisse der `FindDuplicates`-Methode entgegen und erzeugt Dublettengruppen. Dabei wird angenommen, dass die Beziehungen zwischen Dubletten transitiv sind. Ist also laut Ergebnis der Dublettenerkennung $A^{\#}$ eine Dublette von $B^{\#}$ und $B^{\#}$ eine Dublette von $C^{\#}$, dann wird angenommen, dass $A^{\#}$ eine Dublette von $C^{\#}$ ist, selbst wenn die Ähnlichkeit zwischen $A^{\#}$ und $C^{\#}$ unter dem festgelegten Schwellwert liegt. Betrachtet man die Entitäten als Knoten eines Graphen und ein Dublettenpaar als Kante zwischen den beteiligten Entitäten, dann lässt sich das Problem darauf reduzieren in einem unzusammenhängendem Graphen alle zusammenhängenden Teilgraphen zu ermitteln. Der dazu verwendete Algorithmus basiert auf dem Algorithmus von Kruskal zum Finden der minimal spannenden Teilbäume in einem Graph.

Das Resultat ist eine Liste von `DuplicateGroup` Instanzen, die jeweils eine Gruppe repräsentieren. Jede dieser Gruppen enthält eine Liste von Einträgen (`DuplicateGroupEntry`) mit der dazugehörigen Entität. Das Feld `GroupSimilarity` aggregiert das Maximum aller Ähnlichkeitswerte aus allen `DuplicatePair` Paaren an denen die Entität beteiligt war. Der Wert sagt also aus, welche die höchste Ähnlichkeit zwischen dieser

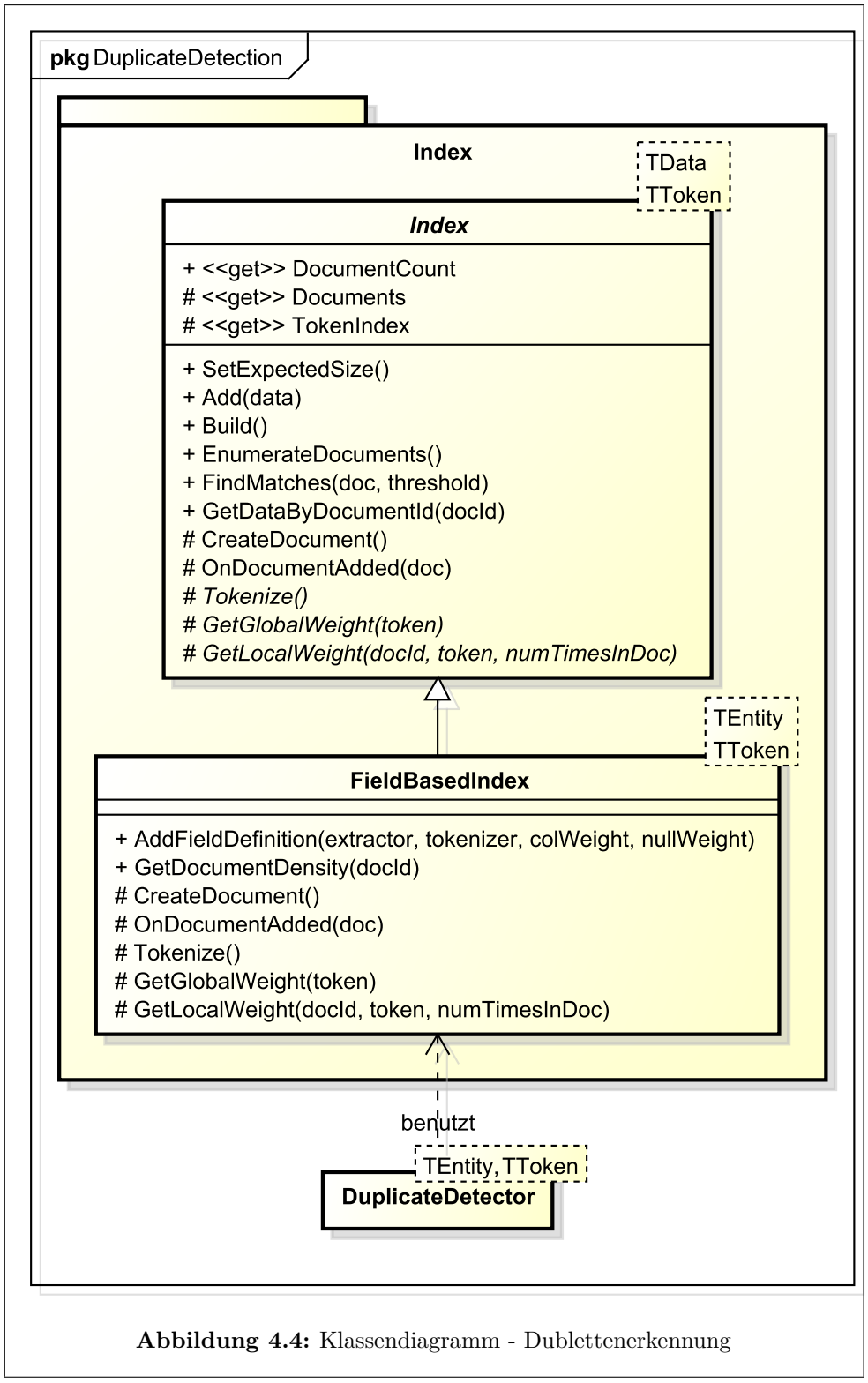


Abbildung 4.4: Klassendiagramm - Dublettenerkennung

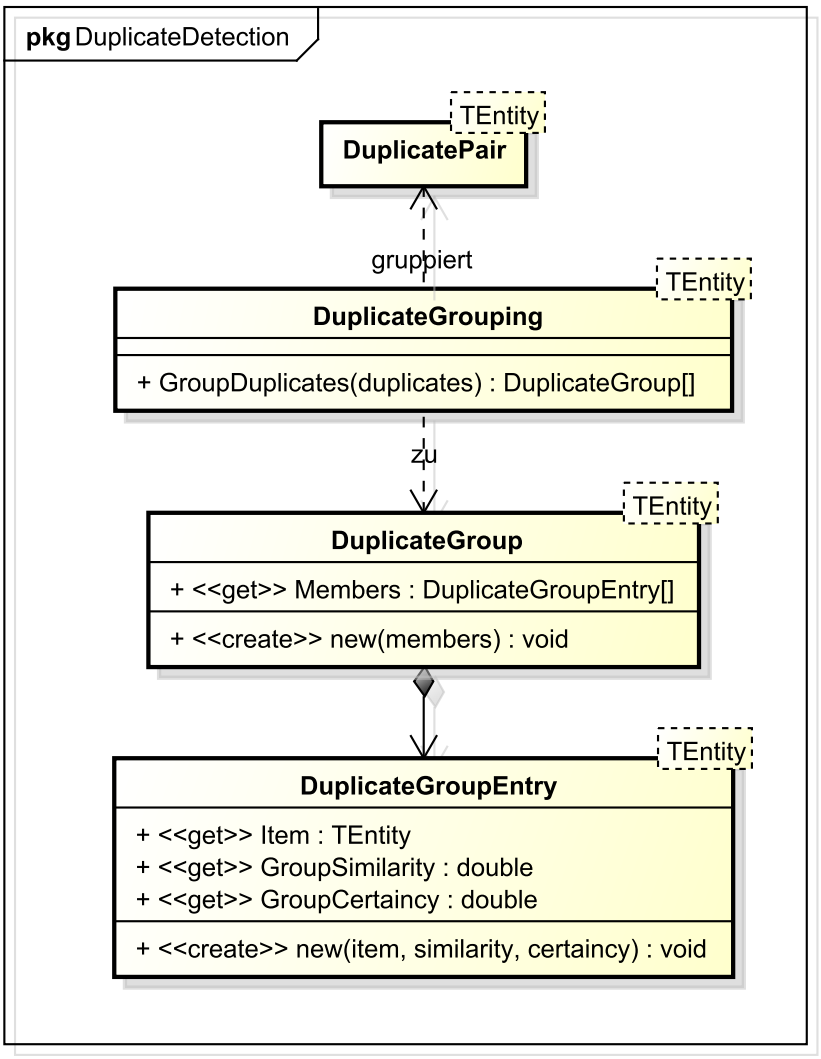


Abbildung 4.5: Klassendiagramm - Gruppierung von Dubletten

Entität und einer anderen Entität der gleichen Gruppe ist. Ähnlich aggregiert das Feld `GroupCertaincy` die „Unsicherheit“ aus allen beteiligten Dublettenpaaren. Hier wird allerdings das Minimum angenommen.

4.3 Webdienst

Der implementierte Webdienst ermöglicht eine einfache Integration der Dublettenerkennung in vorhandene Software. Dazu wird intern die Bibliothek zur Dublettenerkennung benutzt. Der Webdienst übernimmt die Speicherung der Datensätze und Ergebnisse, sowie die Verwaltung von Aufträgen zur Dublettenerkennung.

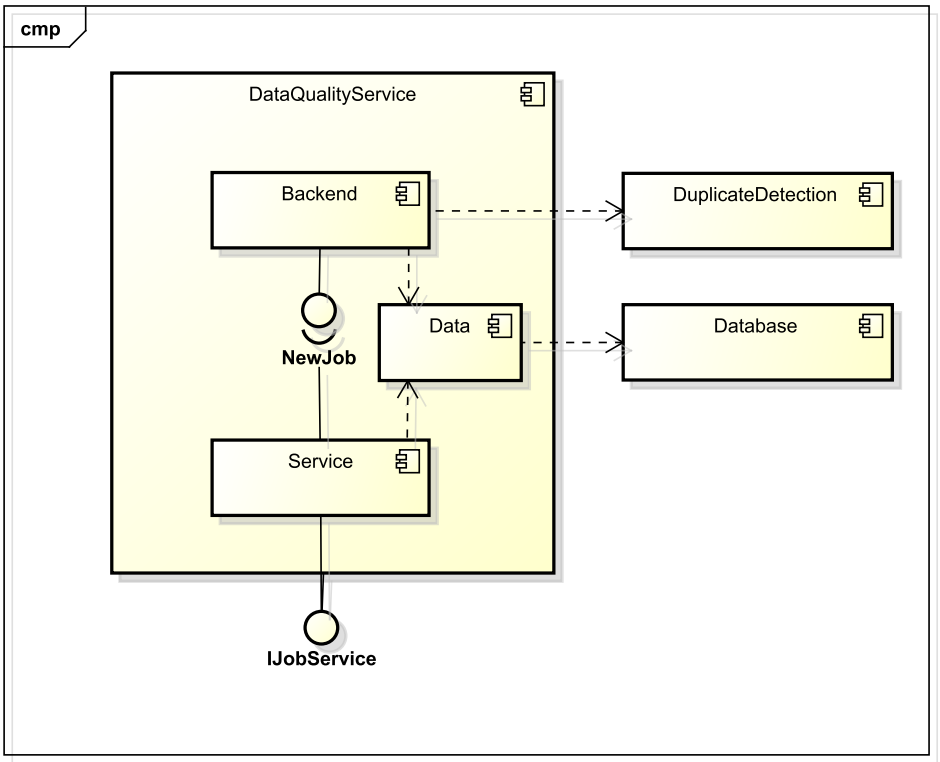


Abbildung 4.6: Komponentendiagramm - DataQualityService

Intern ist der Webdienst in drei separate Komponenten unterteilt. Die eigentliche Dienstschnittstelle befindet sich in der Komponente „Service“. Ihre Aufgabe ist es Anfragen von Clients entgegenzunehmen und zu verarbeiten.

Das Backend des Dublettendienstes ist verantwortlich für das Ausführen von Aufträgen, die über die Dienstschnittstelle übergeben wurden. Die Dienstschnittstelle erstellt dann lediglich eine Entität „Job“ in der Datenbank und benachrichtigt den Client, dass der Auftrag eingestellt wurde und demnächst ausgeführt wird.

Der Zugriff auf die Datenbank erfolgt über die gemeinsam genutzte Komponente `Data`, die über den Objekt-relationalen Mapper (Entity Framework) auf die eigentliche Datenbank zugreift. Die meisten dieser Klassen, sowie die eigentliche Datenbank werden automatisch aus einem Modell generiert und lediglich wo nötig ergänzt.

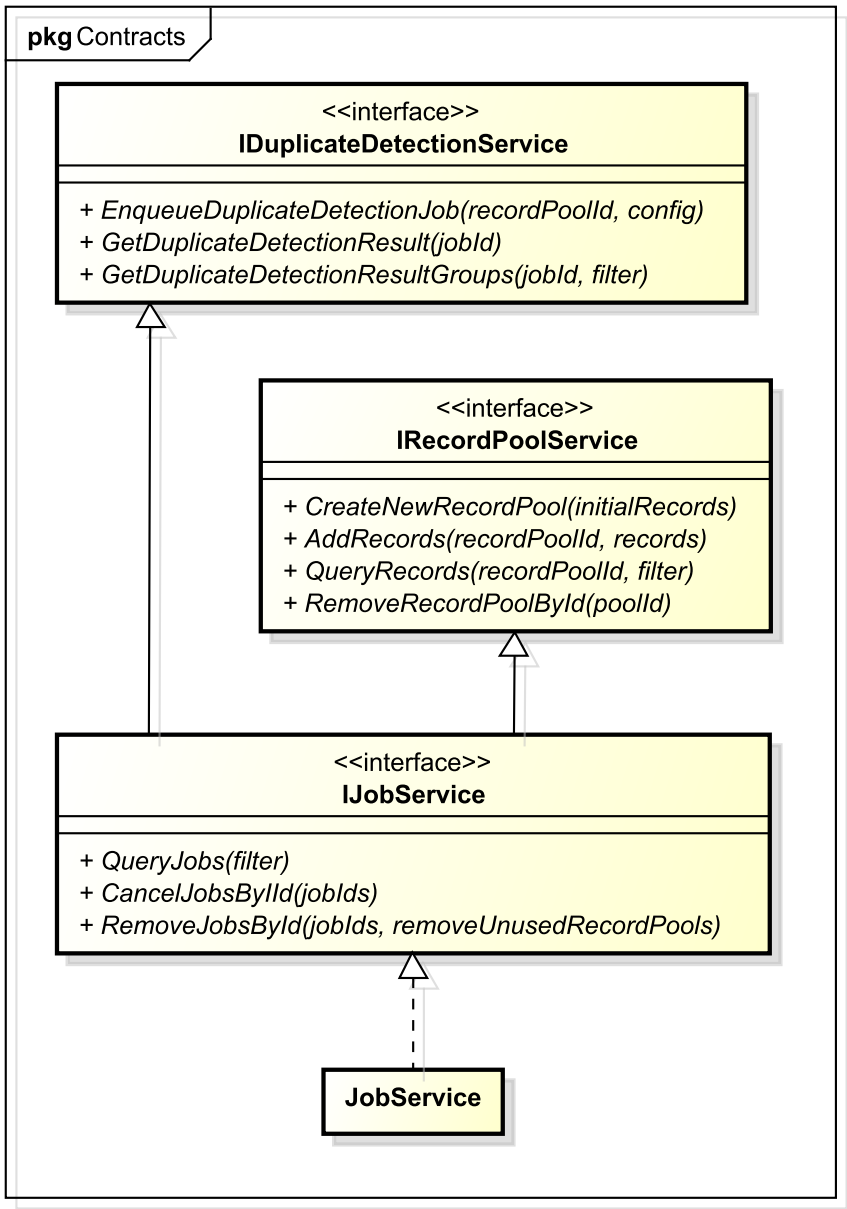


Abbildung 4.7: Klassendiagramm - Dienstverträge

4.3.1 Dienstschnittstelle

Die Dienstschnittstelle ist serverseitig aus drei Schnittstellen zusammengesetzt, die verschiedene Aspekte der Dublettenerkennung beinhalten. Abbildung 4.7 zeigt einige Operationen dieser Schnittstellen.

IRecordPoolService-Schnittstelle

Die Schnittstelle `IRecordPoolService` verwaltet Datensatzmengen. Diese werden zunächst über `CreateNewRecordPool` erstellt. Anschließend können über `AddRecords` weitere Datensätze zu dem Pool hinzugefügt werden. Bei Bedarf können die gespeicherten Datensätze über die Funktion `QueryRecords` wieder abgerufen werden. Um große Datenmengen effizient zu verwalten unterstützt `QueryRecords` – wie auch alle anderen Methoden die möglicherweise größere Datenmengen zurückgeben – das seitenweise Abrufen von Daten („Paging“). Dafür müssen im Parameter `filter` Seitennummer und Seitengröße angegeben werden.

Nach dem eine Datensatzmenge nicht mehr benötigt wird, kann sie über `RemoveRecordPoolById` wieder mit all ihren Datensätzen entfernt werden.

IDuplicateDetectionService-Schnittstelle

Über die Schnittstelle `IDuplicateDetectionService` können Aufträge zur Dublettenerkennung gestartet werden und nach Abschluss die Ergebnisse abgerufen werden. Zum Starten eines Auftrags über `EnqueueDuplicate↙DetectionJob` wird die Kennung der Datensatzmenge benötigt, für die der Auftrag ausgeführt werden soll, sowie Konfigurationsparameter durch die Klasse `DuplicateDetectorConfiguration`, die angeben, wie die Datensätze verarbeitet werden sollen:

Name ermöglicht die Vergabe eines beschreibenden Namens für diesen Auftrag. Dieser kann zum Beispiel direkt vom Benutzer vergeben werden oder auch automatisch generiert werden.

SimilarityThreshold gibt den Schwellwert für die Ähnlichkeit an, die für die Dublettenerkennung verwendet werden soll. Es werden also nur solche Datensätze als Dubletten zurückgegeben, deren Ähnlichkeit größer oder gleich diesem Schwellwert ist.

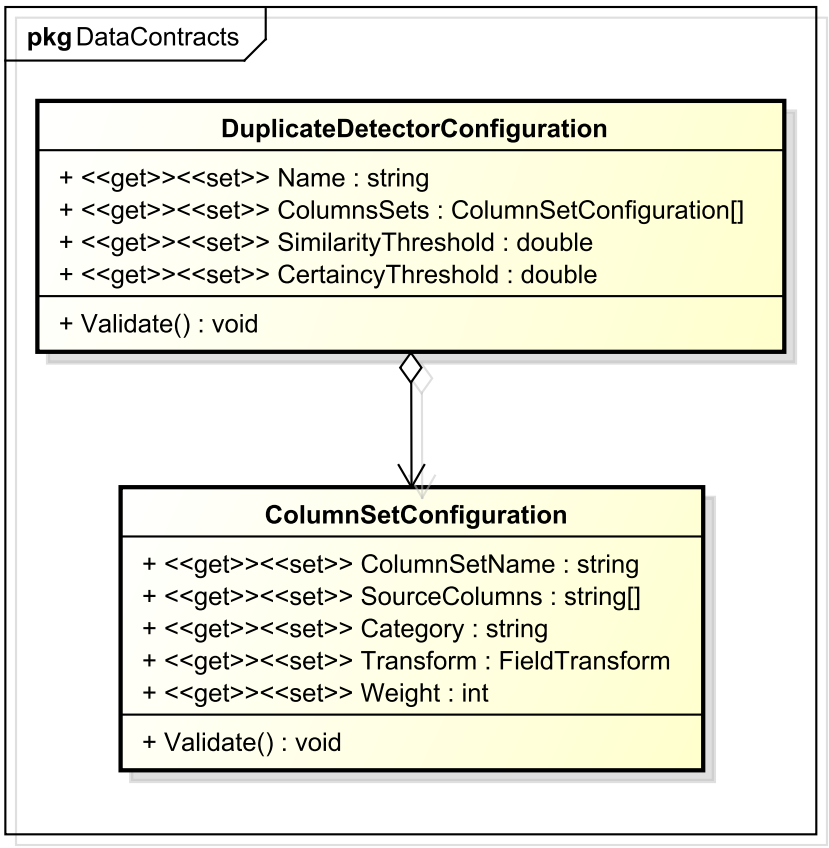


Abbildung 4.8: Klassendiagramm - Konfiguration der Dublettenerkennung

CertaincyThreshold ist ein weiterer Schwellwert, der sich auf die Gewissheit bezieht, mit dem ein Datensatz als Dublette oder nicht-Dublette gekennzeichnet wird. Ist ein Datensatz unvollständig (*NULL*-Werte) so sinkt die Gewissheit. Sinkt sie unter den Schwellwert, so wird der Datensatz nicht als Dublette markiert.

ColumnsSets ermöglicht die Konfiguration der verschiedenen Spalten. Mehrere Spalten können dabei in einem Spaltensatz („ColumnSet“) zusammengefasst werden. Diese werden dann gemeinsam betrachtet, so dass zum Beispiel Feldvertauschungen berücksichtigt werden können. Intern werden dazu die Spalten als ein gemeinsames Feld betrachtet und eine gemeinsame Tokenmenge erzeugt.

Für jeden Spaltensatz können außerdem die folgenden Eigenschaften konfiguriert werden:

ColumnSetName ermöglicht die Vergabe eines beschreibenden Namens für diesen Spaltensatz. Dies dient vor allem diagnostischen Zwecken.

SourceColumns gibt die zu dem Spaltensatz gehörenden Spalten an.

Category kategorisiert den Inhalt der Spalte. Dies dient der automatischen Auswahl einer Stoppwort-Liste, mit der häufige Inhalte entfernt werden können. Für die Kategorie „Straße“ kann zum Beispiel automatisch „str.“, „Straße“, usw. aus dem Feld entfernt werden um dieses weiter zu normalisieren. Dafür muss der Wert `StripCommonWords` für `Transform` angegeben werden.

Stoppwort-Listen werden bisher nicht genutzt, da die tf-idf Gewichtung bereits für eine automatische Abwertung häufig vorkommender Terme sorgt.

Transform gibt an, auf welche Weise die Spalten noch normalisiert werden sollen. Dies ist ein Enum-Wert, der eine Kombination aus `LettersOnly` (alle nicht-Buchstaben entfernen), `IgnoreCase` (Vereinheitlichung der Groß- und Kleinschreibung) und `Strip↙CommonWords` (siehe `Category`).

Weight ist das Gewicht für dieses Feld – relativ zum Gewicht der anderen Felder. Bei höherem Gewicht, hat das Feld mehr Einfluss für die Beurteilung eines Datensatzes als Dublette oder nicht-Dublette.

Nachdem ein Job angelegt und die Ausführung abgeschlossen wurde, können die Resultate über die Funktionen `GetDuplicateDetectionResult` und `GetDuplicateDetectionResultGroups` abgerufen werden. Erstere gibt dabei lediglich die Gesamtzahl der gefundenen Dublettengruppen zurück, während letztere das seitenweise Durchlaufen der Resultatgruppen ermöglicht.

IJobService-Schnittstelle

Über diese Schnittstelle kann der aktuelle Status aller Aufträge mit Hilfe der Funktion `QueryJobs` abgerufen werden. Der `filter` Parameter ermöglicht dabei die Beschränkung auf bestimmte Jobs per Id-Kennung

oder Ausführungsstatus. Der Rückgabewert ist eine Liste von JobInfo-Instanzen, die verschiedene Informationen über die Jobs bereithält.

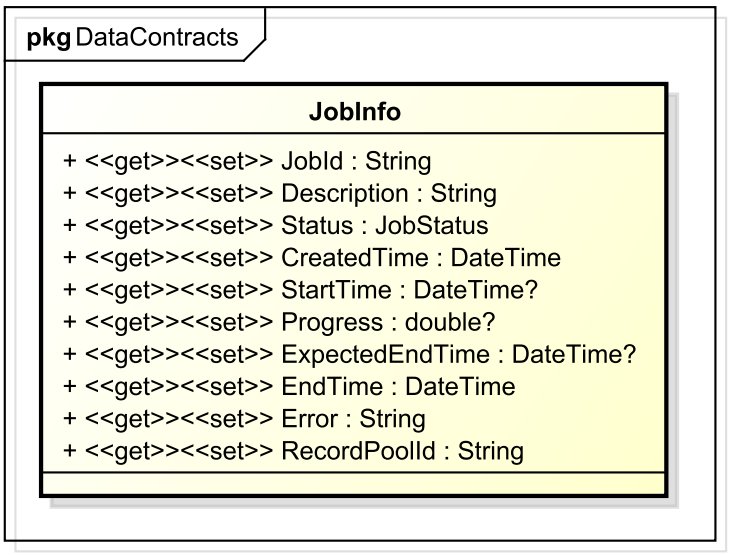


Abbildung 4.9: Klassendiagramm - JobInfo

Die weitere Operationen in der Schnittstelle ermöglichen das Abbrechen eines laufenden Auftrags oder das Löschen eines abgeschlossenen Auftrags aus der Datenbank.

4.3.2 Backend

Das Backend ist verantwortlich für das Ausführen von eingehenden Aufträgen. Es ist so entworfen, dass es möglichst unabhängig von der Dienstschnittstelle arbeitet. Die Kommunikation zwischen Backend und Dienstschnittstelle erfolgt stattdessen primär über die Datenbank. Dies ermöglicht in späteren Versionen die Aufteilung des Webdienstes in separate Ausführungseinheiten für Dienstschnittstelle und Backend und damit auch das Auslagern des Backends auf ein oder mehrere Server, die über eine einzelne Dienstschnittstelle gesteuert werden können.

Weitere Vorteile dieser Architektur sind, dass Zugriffe auf das Backend nicht synchronisiert werden müssen und Fehler in der Dienstschnittstelle das Backend nicht direkt beeinflussen können.

Polling und Synchronisation

Die lose Kopplung zwischen Dienstschnittstelle und Backend bedeutet, dass das Backend keine Information darüber hat, wann ein neuer Job verfügbar ist und auch keine direkte Möglichkeit, die Dienstschnittstelle über den aktuellen Status eines Auftrags zu informieren.

Dieses Problem lässt sich durch Polling bzw. periodisches synchronisieren lösen. Dabei prüft das Backend, durch einen Zeitgeber bestimmt (z.B. alle 5 Sekunden), ob neue Jobs zur Verarbeitung bereitstehen und führt diese als asynchrone Tasks aus. Während der Ausführung werden Statusinformationen, wie der aktuelle Fortschritt und das voraussichtliche Abschlussdatum, ebenfalls periodisch in die Datenbank geschrieben. Diese können von den Clients über die Dienstschnittstelle abgerufen werden.

Da das Backend nur in festgelegten Zeitabständen den Zustand übermittelt und unabhängig von der Dienstschnittstelle agiert, wird es durch übermäßige Anfragen des Frontends nicht überlastet. Die Anfragen an die Dienstschnittstellen können zudem durch Mechanismen im Datenbankserver oder über einen speziellen Cache-Server zwischengespeichert werden.

Für den Fall, dass ein Server gerade keine Aufträge ausführt und auch keine neuen Aufträge übermittelt werden, führt das Polling zu unnötigen Abfragen an die Datenbank. Um diesen Effekt abzumildern, wurde eine einzelne Optimierung eingeführt, um das Backend mit Hilfe eines Signals (`AutomaticResetEvent`) zu benachrichtigen, wenn ein neuer Job über die Schnittstelle eingestellt wurde. Der Zeitgeber wird dann nur aktiv, wenn entweder ein neuer Auftrag eingestellt wurde oder sich ein Auftrag in Ausführung befindet. Auch in diesem Fall wird aber, innerhalb des Zeitintervalls, höchstens einmal eine Datenbanksynchronisation durchgeführt.

Die verwendeten Signale lassen sich auch prozessübergreifend nutzen, wenn eine Aufteilung von Backend und Dienstschnittstelle gewünscht ist. Sollen beide Komponenten auf unterschiedlichen Rechnern laufen, so müssen andere Signalisierungsmechanismen oder ein reines Polling verwendet werden.

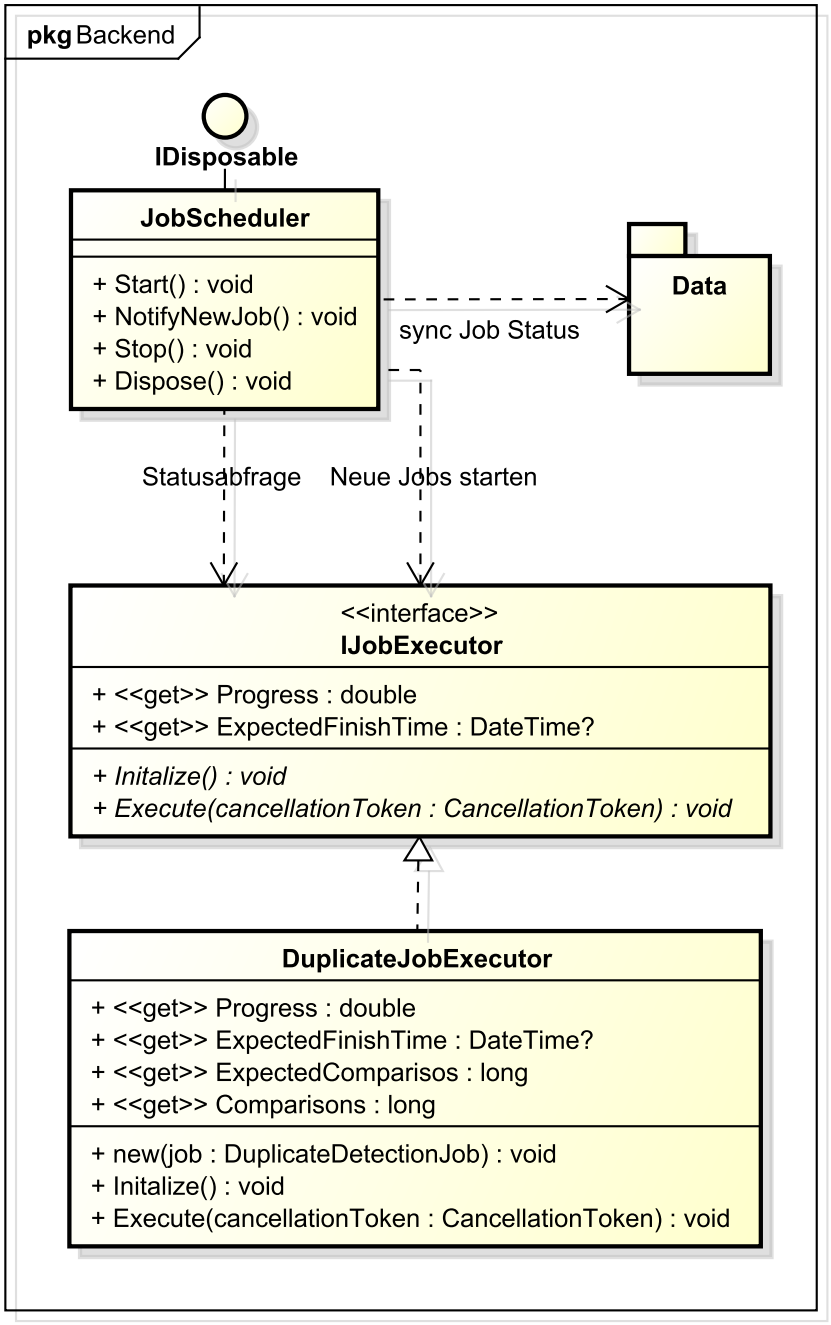


Abbildung 4.10: Klassendiagramm - Backend

Struktur

Kern des Backends ist die Klasse `JobScheduler`. Der Aufruf der Funktion `Start` erstellt einen neuen Hintergrundtask, der über die Funktion `Stop` beendet werden kann. Der Aufruf der Funktion `Stop` sorgt auch für das automatische Abbrechen aller Aufträge, die von dem `JobScheduler` verwaltet werden und ermöglicht damit ein sauberes Herunterfahren des Servers. Die Funktion `NotifyNewJob` setzt intern ein Signal vom Typ `AutomaticResetEvent`, mit dem das Backend über neue Aufträge benachrichtigt werden kann.

Sobald ein neuer Auftrag aus der Datenbank gelesen wurde, erstellt der `JobScheduler` einen für den Auftrag passenden `IJobExecutor`. Für die Dublettenerkennung ist dies eine Instanz der Klasse `Duplicate↙ JobExecutor`. Die Schnittstelle `IJobExecutor` bietet Methoden für die Ausführung der Aufträge, sowie Eigenschaften mit denen der aktuelle Status der Ausführung abgerufen werden kann. Das aus dem .NET-Framework stammende `CancellationToken` ermöglicht ein threadsicheres Abbrechen eines Auftrags. Der `JobScheduler` führt die `Execute`-Methode in einem neuen Hintergrundthread aus.

4.4 Datenbank

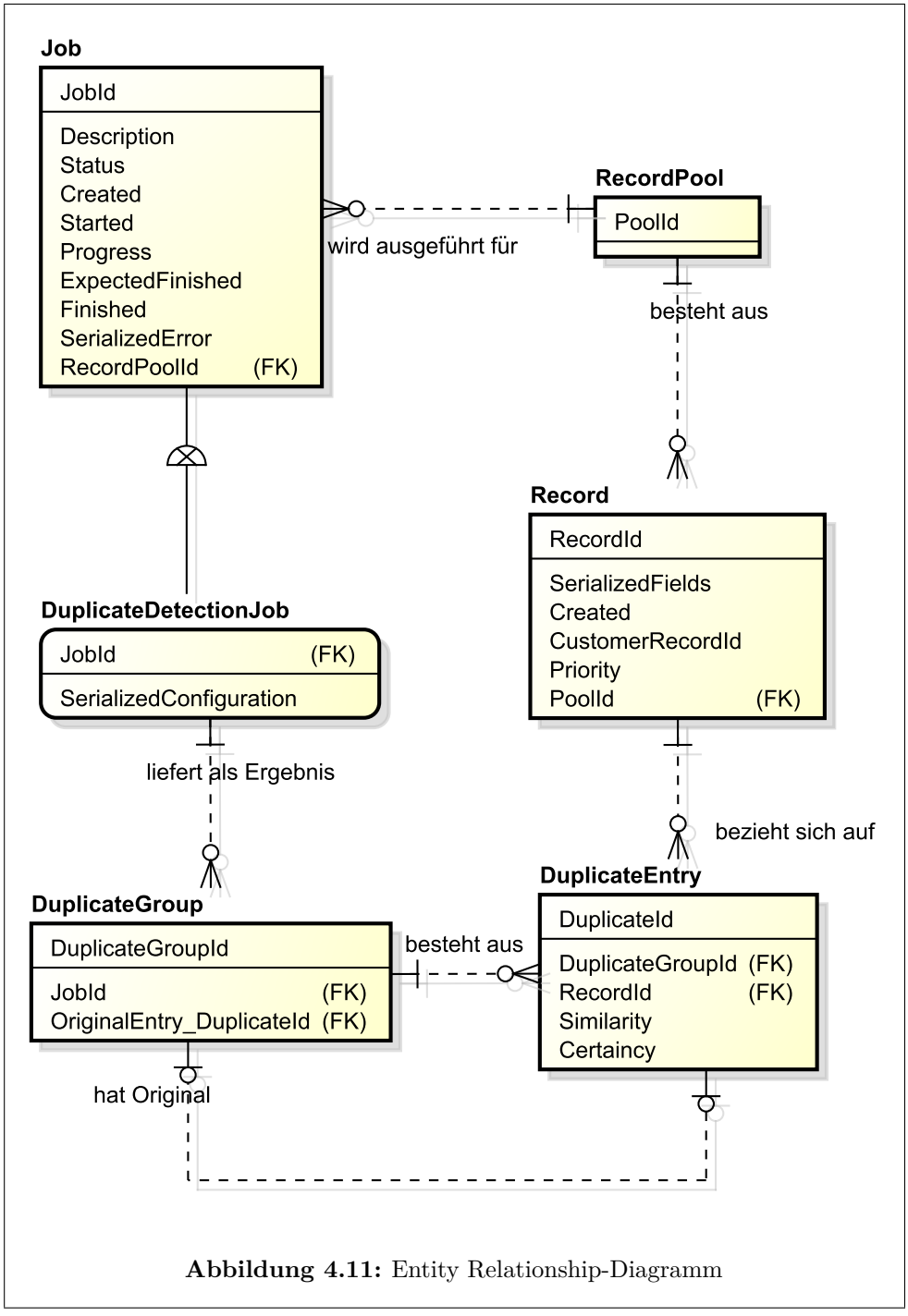


Abbildung 4.11: Entity Relationship-Diagramm

Abbildung 4.11 zeigt einen Ausschnitt aus dem verwendeten ERM (Entity Relationship Modell) in Krähenfußnotation mit den wichtigsten für die Dublettenerkennung verwendeten Entitäten und den dazugehö-

rigen Attributen. Diese werden in den folgenden Abschnitten genauer beschrieben.

4.4.1 Entität Job

Diese Entität repräsentiert eine auszuführende, sich in Ausführung befindende oder bereits ausgeführte Aufgabe im System. Alle im System existierenden Aufgabentypen sind Subtypen dieser Entität. Für die Dublettenerkennung interessant ist dabei der Subtyp `DuplicateDetectionJob`.

Neue zu erledigende Aufgaben werden vom Webdienst in dieser Entität gespeichert und dann vom Backend verarbeitet.

Description ist eine Bezeichnung für den Job, die vom Client beziehungsweise dem Benutzer vergeben werden kann.

Status ist ein Enumerationswert der den aktuellen Status der Aufgabe wiedergibt. Nach dem Anlegen einer neuen Aufgabe durch den Webdienst befindet sich die Aufgabe im Status 1 (`Created`). Sobald das Backend die Aufgabe geladen hat, ändert es den Status auf 2 (`Pending`) und startet einen Hintergrundprozess in einer Warteschlange um die Aufgabe zu verarbeiten. Der Hintergrundprozess ändert den Status auf 4 `Executing`) um zu kennzeichnen, dass die Aufgabe sich in Ausführung befindet. Ab diesem Zeitpunkt wird in regelmäßigen Abständen der aktuelle Fortschritt in die Datenbank geschrieben (siehe `Progress`). Nachdem ein Job erfolgreich abgeschlossen ist, wird der Status auf den Wert 16 (`Processed`) geändert. Zu diesem Zeitpunkt können die Ergebnisse über die verknüpften Entitäten `DuplicateGroup` und `DuplicateEntry` abgerufen werden. Im Falle eines Fehlers während der Ausführung wird der Wert auf 32 (`Error`) geändert. Die Werte 8 (`Cancelling`) und 64 (`Cancelled`) werden zum Abbrechen eines Jobs während der Ausführung benötigt. Die Verwendung von Zweier-Potenzen erleichtert die Abfrage von Aufgaben mit bestimmten Statuswerten über eine Maske. Die gültigen Werte werden in der Enumeration `JobStatus` verwaltet.

Created speichert den Zeitpunkt, zu dem die Aufgabe erstellt wurde.

Started speichert den Zeitpunkt, zu dem die Aufgabe gestartet wurde. Dies entspricht dem Zeitpunkt, zu dem der Status auf `Executing` geändert wurde.

Progress enthält den aktuellen Fortschritt der Aufgabe als Fließkommazahl zwischen 0 (0%) und 1 (100%). Der aktuelle Fortschritt wird vom Backend regelmäßig mit der Datenbank synchronisiert und kann über den Webdienst abgerufen werden.

ExpectedFinished gibt Auskunft darüber, zu welchem Zeitpunkt die Aufgabe voraussichtlich abgeschlossen sein wird.

Finished ist der tatsächliche Zeitpunkt, zu dem eine Aufgabe abgeschlossen wurde.

SerializedError speichert gegebenenfalls aufgetretene Fehler während der Ausführung. Es handelt sich dabei um als XML serialisierte Daten der Klasse `ExceptionInfo`, die eine Ausnahme kapselt.

RecordPoolId ist ein Verweis auf die Datensatzmenge (Entität `RecordPool`), für die der Job ausgeführt werden soll.

4.4.2 Entität RecordPool

Die Entität `RecordPool` kapselt eine Datensatzmenge, für die Aufträge (Jobs) erstellt werden können.

4.4.3 Entität Record

Die Entität `Record` stellt einen einzelnen Datensatz dar, der Teil einer Datensatzmenge (`RecordPool`) ist. Da über den Webdienst beliebige Eingabeformate unterstützt werden, sind die verwendeten Spalten im Vorhinein nicht bekannt und können daher auch nicht in der Tabelle `Record` definiert werden. Eine spezielle Attribut-Tabelle mit Schlüssel-Wert-Paaren wäre möglich, würde aber die Abfrage der Daten erschweren und möglicherweise verlangsamen. Da die Felddaten ausschließlich im Backend verarbeitet werden, würde eine solche Vorgehensweise auch wenig Vorteile bieten. Stattdessen werden die Attribute in serialisierter Form in dem Feld `SerializedFields` gespeichert.

Created speichert einen Zeitstempel, zu dem der Datensatz angelegt wurde.

CustomerRecordId ist ein vom Client definierter, zusätzlicher Wert der den Datensatz eindeutig identifiziert. Dies erleichtert die automatische Verarbeitung der Ergebnisse.

Priority gibt die Datensatzpriorität an. Diese wird im Falle der Dublettenerkennung verwendet, um bei der Erstellung einer Dublettengruppe automatisch einen Kandidaten auszuwählen, der als „Original“ gekennzeichnet wird. Niedrigere Werte bezeichnen eine höhere Priorität.

PoolId ist ein Verweis auf die Datensatzmenge, zu dem dieser Datensatz gehört.

4.4.4 Entität DuplicateDetectionJob

Diese Entität ist ein Untertyp der Entität `Job` und speichert Aufträge zur Dublettenerkennung in einer bestimmten Datensatzmenge. Die Konfigurationsparameter für die Dublettenerkennung werden in dem Feld `SerializedConfiguration` gespeichert. Dies ist eine als XML serialisierte Instanz der Klasse `DuplicateDetectorConfiguration`, die dem Webdienst als Parameter übergeben wurde.

4.4.5 Entität DuplicateGroup

Nach Abschluss der Dublettenerkennung werden alle gefundenen Dubletten gruppiert und als `DuplicateGroup`-Entitäten gespeichert. Jede dieser Gruppen enthält ein oder mehrere `DuplicateEntry`-Entitäten, die die Mitglieder dieser Gruppe beschreiben.

Eines der Mitglieder wird als Original gekennzeichnet. Die Unterscheidung zwischen Original und Dublette ist in der Praxis oft wichtig, da die Datensätze im Client gegebenenfalls zusammengeführt werden müssen oder Dubletten gelöscht werden. In diesem Fall muss klar sein, welcher der Einträge als Original behandelt werden soll. Standardmäßig ist dies der Eintrag, dessen zugehöriger Datensatz die höchste Priorität hat (Attribut `Priorität` in Entität `Record`). Welcher Eintrag als Original behandelt werden soll, kann über den Webdienst geändert werden.

4.4.6 Entität DuplicateEntry

In `DuplicateEntry` werden die Mitglieder einer bestimmten Dublettengruppe (`DuplicateGroup`) verwaltet.

DuplicateGroupId bezeichnet die Gruppe, zu der der Eintrag gehört.

RecordId verweist auf den Datensatz, der als Dublette identifiziert wurde.

Similarity gibt an, welche Ähnlichkeit dieser Datensatz zu den anderen Datensätzen in dieser Gruppe besitzt. Mögliche Werte liegen zwischen 0 und 1 und sind durch das verwendete Ähnlichkeitsmaß definiert.

Certaincy gibt an, welche Aussagekraft die Angabe zur Ähnlichkeit hat. Dieser Wert ist ebenfalls von dem verwendeten Ähnlichkeitsmaß abhängig. Bei dem hier verwendeten Algorithmus bezeichnet es, wie vollständig der dazugehörige Datensatz ist, also welcher Anteil des Datensatzen aus *NULL*-Werten besteht.

4.5 Clients

Zurzeit existieren zwei Clients die den Dienst zur Dublettenerkennung nutzen können. Diese wurden in Zusammenarbeit mit der Firma *codegarden software* und weiteren Personen entwickelt.

4.5.1 DataQualityServices.de

Die Webseite *DataQualityServices.de* stellt einen Silverlight basierenden Client zur Verfügung, der es erlaubt CSV Dateien auf Dubletten zu überprüfen. Die Daten werden dazu an einen Server übermittelt, auf dem der in diesem Kapitel beschriebene Webdienst installiert wurde.

Nach dem Auswählen einer CSV Datei können verschiedene Konfigurationsparameter festgelegt und der Auftrag gestartet werden. Die Ergebnisse lassen sich über die Oberfläche betrachten oder als CSV Datei exportieren.

4.5.2 Kontor .NET

Kontor .NET ist ein Warenwirtschaftssystem von *codegarden software*.

Ein AddOn ermöglicht es, Adressdaten an den Dienst zu übermitteln und diese auf Dubletten zu überprüfen. Die Ergebnisse können dann direkt in der Warenwirtschaft weiterverarbeitet werden.

5 Leistungsbewertung

Zur Beurteilung der Leistungsfähigkeit des Systems wird die Relevanz der Ergebnisse sowie die benötigte Laufzeit betrachtet.

Eine hohe Relevanz bei der Duplikatenerkennung ergibt sich, wenn die Anzahl der gefundenen, echten Duplikate möglichst hoch ist („Trefferquote“), während möglichst wenig Datensätze fälschlicherweise als Duplikate erkannt werden sollten („Genauigkeit“). In der englischsprachigen Literatur sind diese Faktoren als „recall“ und „precision“ bekannt.

Es ist leicht ersichtlich, dass beide Faktoren voneinander abhängig sind und bei steigender Genauigkeit die Trefferquote im Allgemeinen sinkt. Ein System, das einfach jeden Datensatz als Duplikat markiert, wird tatsächlich alle vorhandenen Duplikate "erkennen" und hätte eine hundert prozentige Trefferquote. Der größte Anteil der Ergebnisse wird aber irrelevant sein und die Präzision entsprechend gering. Ein anderes System, welches Duplikate nur bei absoluter Sicherheit als solche markiert, wird dagegen eine hohe Genauigkeit haben, aber möglicherweise viele Duplikate übersehen und damit eine geringe Trefferquote haben.

Je nach Anwendungsfall kann eine hohe Trefferquote oder eine hohe Genauigkeit als wichtiger empfunden werden, weshalb es sinnvoll ist, dies für den Benutzer konfigurierbar zu machen. In dem hier vorgestellten System wird dies über die Einstellung des Schwellwerts ermöglicht, ab dem ein Datensatz als Duplikat eines anderen Datensatzes erkannt wird.

5.1 Testdaten

Die Evaluation der oben genannten Faktoren benötigt eine geeignete Auswahl an Testdaten. Im Idealfall sollte es sich dabei um authentische Daten aus der Anwendungspraxis handeln. Zur Betrachtung der Relevanz ist außerdem wichtig, dass alle tatsächlich vorhandenen Duplikate bekannt sind, denn nur so können Trefferquote und Genauigkeit akkurat berechnet werden. In kleinen Datenbeständen lässt sich dies durch ma-

nuelle Durchsicht aller Datensätze erreichen. In großen Datenbeständen ist dies praktisch nicht mehr möglich. Da es sich bei Adressbeständen zu dem in der Regel um vertrauliche Daten handelt, sind diese nur schwer zugänglich und nicht zur Veröffentlichung geeignet, wodurch Ergebnisse später nicht reproduzierbar sind.

Eine Alternative stellen selbst generierte Daten dar, die in ihrer Struktur die Realität möglichst gut abbilden sollten. Da in diesem Fall auch die Dubletten generiert werden, ist genau bekannt, welches die Dubletten im Datenbestand sind und Trefferquote und Genauigkeit lassen sich exakt bestimmen. Diese Alternative wird hier für die Evaluierung des Algorithmus genutzt.

Die resultierende Lösung wurde zudem an weiteren Datenbeständen getestet, die reale Adressdaten enthalten, um sicherzustellen, dass die Ergebnisse auch auf die Praxis übertragbar sind. Da hier allerdings nicht im Vorhinein bekannt war, welche Dubletten sich im Datenbestand finden, können diese lediglich einen subjektiven Eindruck über die Relevanz der Ergebnisse bieten.

5.1.1 Generierung der Adressen

Bei der Generierung der Testdaten wurde darauf geachtet, möglichst natürliche Adressen zu generieren. Dazu wurden Statistiken aus mehreren Quellen zusammengetragen.

Zur Generierung von Vornamen wurden z.B die 1000 häufigsten männlichen und die 1000 häufigsten weiblichen deutschen Vornamen aus einem Telefonbuch benutzt. Diese fanden sich in Balû (2009) und Balû (2005b). Eine ähnliche Liste existiert auch für die 1000 häufigsten Nachnamen in Deutschland (vgl. Balû 2005a).

Alle diese Listen enthalten die absolute Häufigkeit mit denen die Namen im Telefonbuch vorkamen. Aus diesen Informationen lassen sich also Paare aus Vor- und Nachnamen generieren, die der statistischen Verteilung der Originaldaten entsprechen.

Für die Generierung der Anschrift wurden Postleitzahlen und die dazugehörigen Ortsnamen aus öffentlich zugänglichen Quellen kopiert. Eine Liste von etwa 7500 Straßennamen wurde mit einem Web-Crawler generiert. Jedem Postleitzahlbereich wurde dann per Zufall eine Liste von etwa 100 Straßennamen zugeordnet, was – durch Stichproben ermittelt – einer üblichen Straßenzahl innerhalb eines Postleitzahlbereiches entspricht.

Auch Email-Adressen werden automatisch generiert, sind aber abhängig von dem jeweiligen Namen. Dazu wird nach dem Zufallsprinzip ein Adressschema und weitere Daten, wie der Provider ausgewählt und anhand des Schemas und des Namens eine Email-Adresse generiert. Ein typischen Schema wäre z.B. [Vorname]-[Nachname]@[Provider].

Hier sind einige Beispiele für Adressen die damit generiert wurden:

Frau Valeria Haag
Bärensteiner Straße 23
2991 Leippe-Torno
valeria.haag@yahoo.de

Frau Beatrice Schmieder
Trachauer Straße 7
55758 Asbach
info@schmieder.de

Herr Jochen Wiemann
Südhöhe 6
64319 Pfungstadt
jochenw@gmx.net

5.1.2 Generierung der Dubletten

Neben den eigentlichen Adressdaten müssen ebenfalls Dubletten erzeugt werden. Um die Realität auch hier in etwa nachzubilden, sollte es sich dabei nicht um identische Datensätze handeln, sondern um Datensätze die in verschiedenen Aspekten vom Original abweichen.

Dazu wurde ein „Mutierer“ entwickelt, der aus einem Originaldatensatz eine abgeänderte Dublette erzeugt. Hierbei kommen nach Zufallsprinzip verschiedene Techniken zum Einsatz die einige häufige Datenfehler in der Realität reproduzieren sollen:

- Einfügen, entfernen und vertauschen einzelner Buchstaben in einem zufälligen Adressfeld.
- Vertauschen von Vorname und Nachname.
- Geänderte Postleitzahl in der letzten Stelle.
- Abweichende Email-Adresse.

Jede Dublette enthält mindestens einer dieser Abweichungen. Mit einer Wahrscheinlichkeit von 50% wird die Dublette erneut mutiert. Diese erneut mutierte Adresse wird ebenfalls wieder mit einer Wahrscheinlichkeit von 50% erneut mutiert (usw.).

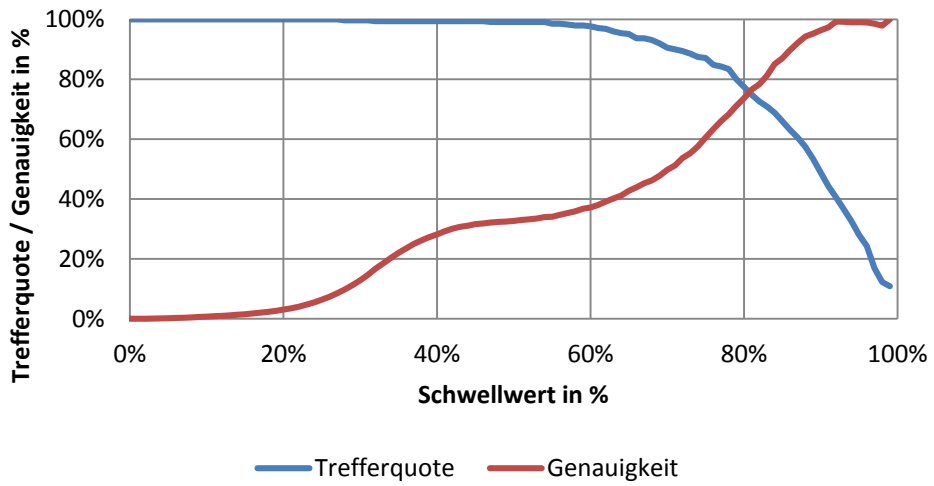
Die Erfahrung aus echten Testdaten zeigt zudem, dass im gleichen Haushalt lebende Personen häufig fälschlicherweise als Dubletten markiert werden, wenn der Schwellwert nicht hoch genug ist, weil hier Anschrift und oft sogar der Nachname identisch sind. Der Anteil solcher Adressen ist nach eigener Beobachtung dabei oft größer, als der Anteil der tatsächlichen Dubletten. Damit haben diese Adressen einen großen Einfluss auf die Erkennungsleistung und die Wahl des Schwellwerts.

Um diesen Faktor ebenfalls zu berücksichtigen werden neben den Dubletten auch explizite nicht-Dubletten erzeugt, die allerdings die gleiche Anschrift wie eine andere Adresse aus dem Datenbestand besitzen. Dabei wird eine vorhandene Adresse aus dem Datenbestand entnommen und nur der Vorname und die Email-Adresse ausgetauscht.

Der Anteil der Dubletten im Datenbestand Ψ wird im folgenden als d bezeichnet und der Anteil an ähnlichen Adressen im gleichen Haushalt, die aber keine Dubletten sind als h.

5.2 Relevanz

Zur Beurteilung der Relevanz betrachten wir zunächst die Abhängigkeit von Trefferquote und Genauigkeit vom Schwellwert. Wie in Abbildung 5.1 zu erkennen ist, steigt die Genauigkeit bei höherem Schwellwert, gleichzeitig sinkt aber ab einem gewissen Punkt die Trefferquote.



$\|\Psi\| = 1000, d = 5\%, h = 10\%$, Werte gemittelt (4 Durchläufe)

Abbildung 5.1: Trefferquote und Genauigkeit in Abhängigkeit vom Schwellwert

Das Plateau bei etwa 50% ergibt sich aus den vorher angesprochenen ähnlichen Datensätzen, also den Adressen, die aus dem gleichen Haushalt stammen. Da diese Adressen eine hohe Ähnlichkeit zu den anderen Adressen aufweisen, stabilisieren sich an dieser Stelle die Ergebnisse ein wenig, bis der Schwellwert ausreichend steigt, um zwischen ähnlichen Adressen und tatsächlichen Dubletten besser zu unterscheiden. Es zeigt sich aber auch, dass ab diesem Punkt die Trefferquote allmählich absinkt, da hierdurch auch einige tatsächliche Dubletten übersprungen werden.

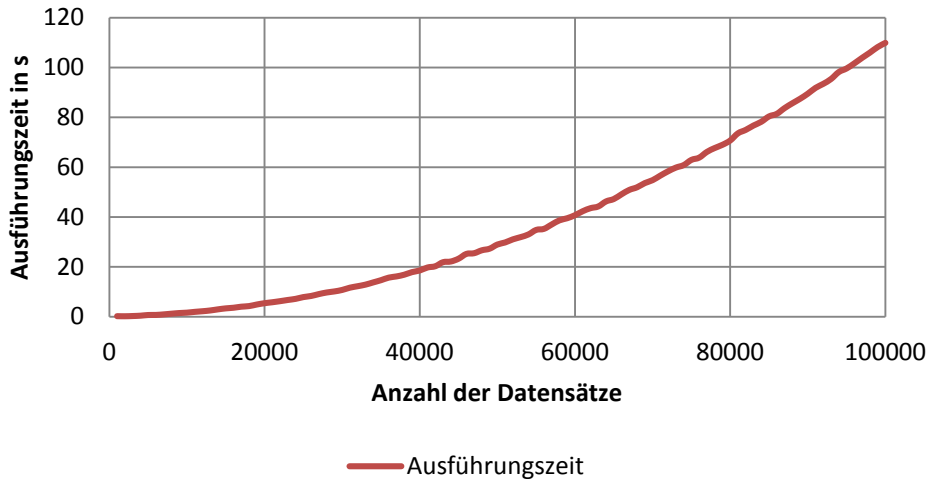
Ein ausgeglichenes Verhältnis zwischen Trefferquote und Genauigkeit ergibt sich in diesem Beispiel bei etwa 80%. Die Erfahrung zeigt, dass bei größeren Datenbeständen ein noch höherer Schwellwert gewählt werden sollte, da bei zunehmender Bestandsgröße der Unterschied zwischen den einzelnen Datensätzen geringer wird[1] und damit die Genauigkeit abnimmt.

[1] Der Hamming-Abstand zwischen den einzelnen Datensätzen sinkt.

5.3 Laufzeit

5.3.1 Laufzeit in Abhängigkeit von der Eingabegröße

Neben der Relevanz der Ergebnisse ist natürlich die Laufzeit des Algorithmus für verschiedene Eingabegrößen interessant. In Abbildung 5.2 wurde die Laufzeit für verschiedene Eingabegrößen ermittelt.



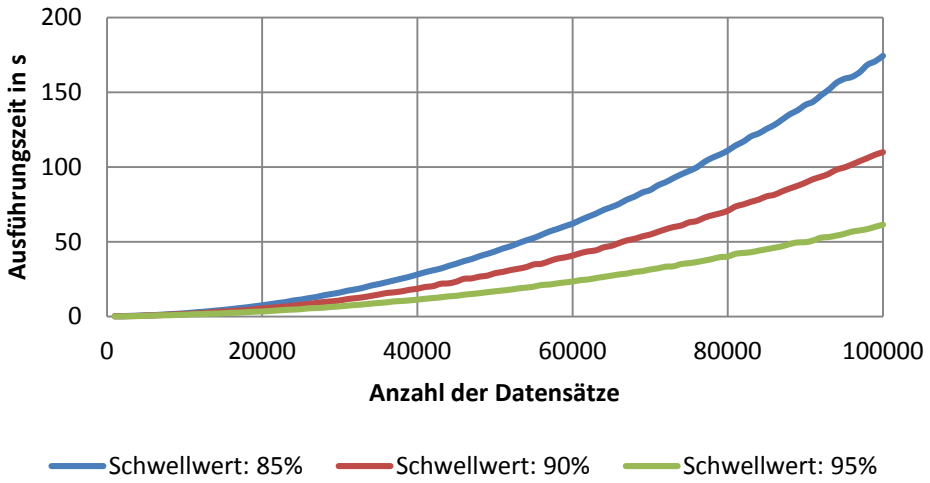
$\sigma = 90\%, d = 5\%, h = 10\%$

Abbildung 5.2: Tatsächliche Laufzeit in Abhängigkeit zur Anzahl der Datensätze

5.3.2 Einfluss des Pruning

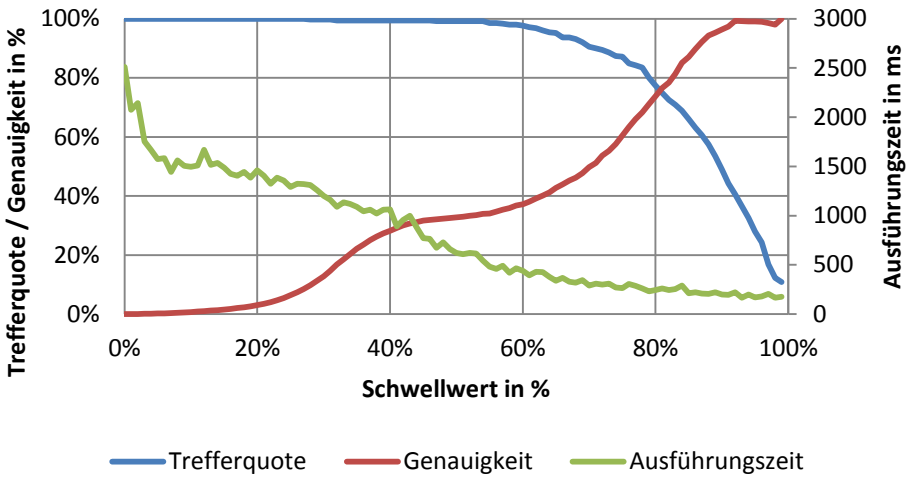
Die im Kapitel „**Entwurf**“ vorgestellte Optimierung der Dublettensuche mit Hilfe von Pruning wirkt sich deutlich auf die Laufzeit der Dublettenerkennung aus. Die Beschleunigung ist dabei direkt abhängig von dem festgelegten Schwellwert. Ein höherer Schwellwert sorgt dafür, dass viele Adressen vorzeitig ausgeschlossen werden können und damit die Dublettenerkennung schneller beendet wird.

Abbildung 5.3 zeigt drei verschiedene Schwellwerte im Vergleich. In Abbildung 5.4 wird die Laufzeit in Abhängigkeit vom Schwellwert bei fester Eingabegröße dargestellt. Man kann sehen, dass die Laufzeit sogar überproportional vom Schwellwert abhängt. Gerade größere Datenbestände profitieren davon, da hier in der Regel schon wegen der höheren Genauigkeit größere Schwellwerte benutzt werden.



$d = 5\%, h = 10\%$

Abbildung 5.3: Tatsächliche Laufzeit in Abhängigkeit zur Anzahl der Datensätze mit verschiedenen Schwellwerten



$\|\Psi\| = 1000, d = 5\%, h = 10\%$, Werte gemittelt (4 Durchläufe)

Abbildung 5.4: Laufzeit in Abhängigkeit vom Schwellwert

6 Ausblick und Zusammenfassung

Das vorgestellte Projekt zeigt, wie die klassische Dublettenerkennung von modernen Technologien aus dem Suchmaschinenbereich und der gewachsenen Parallelisierung profitieren kann und wie sich ein solches System implementieren lässt, so dass es möglichst einfach in bestehende Infrastruktur eingebunden werden kann.

Dennoch gibt es viele Möglichkeiten, das bestehende System in der Zukunft zu erweitern und zu verbessern. Einige Ideen sollen hier zusammengefasst werden.

6.1 Flexiblerer Einsatz

Ein wichtiges Gebiet für die Dublettenerkennung ist das Zusammenführen mehrerer Datenbestände aus verschiedenen Quellen, da es hier leicht zur Einführung von Dubletten kommen kann. Diese Variante wird durch die hier vorgestellte Implementierung nicht gut unterstützt. Es gibt zwar die Möglichkeit, alle Datensätze zu einer gemeinsamen Datenmenge zu vereinen und danach eine Dublettenerkennung mit allen Datensätze durchzuführen, effizienter wäre es aber, wenn lediglich der neue Bestand gegen den alten Bestand geprüft wird.

Den Extremfall stellt das Prüfen eines einzelnen Datensatzes gegen den bestehenden Datenbestand dar. Auch das macht Sinn, wenn zum Beispiel bereits beim Eingeben neuer Adressen geprüft werden soll, ob es sich um eine Dublette handelt. Für diese Fälle könnte die Implementierung um weitere Methoden ergänzt werden, die diese Szenarien besser unterstützen. Die im Kapitel „**Entwurf**“ vorgestellten Techniken lassen sich dabei ohne größere Änderungen auf diese Anwendungsfälle übertragen, so dass lediglich die Implementierung angepasst werden müsste.

6.2 Permanente Indizierung

Eng mit dem letzten Vorschlag verbunden ist die Einführung eines permanenten Index. Während zurzeit der Index für jede Dublettenerkennung neu erstellt wird und einen nicht unerheblichen Teil der gesamten Bearbeitungszeit ausmacht, so könnte in Zukunft der Index, wie es in anderen Datenbanksystemen üblich ist, persistent gespeichert und wiederverwendet werden.

Die Auslagerung des Index auf externe Medien ermöglicht zudem eine noch höhere Skalierbarkeit, die nur noch durch den Speicherplatz auf der Festplatte beschränkt wird. Da der Zugriff auf die Festplatte jedoch sehr langsam ist, sollte versucht werden den kompletten Index im Hauptspeicher zu halten. Beim Starten der Dublettenerkennung könnte geprüft werden, ob sich der Index bereits im Speicher befindet und – wenn das nicht der Fall ist – komplett von der Festplatte geladen werden. Dazu lassen sich diverse Techniken nutzen, um den Index zu komprimieren und damit den Hauptspeicher zu entlasten. Eine sehr gute Übersicht zu verschiedenen Komprimierungstechniken findet sich in Büttcher, Clarke und Cormack (2010, Kapitel 6).

Der bisherige Index ist zudem statisch, das bedeutet, dass er einmalig aus allen bekannten Dokumenten erstellt wird. Kommen Datensätze hinzu oder werden gelöscht, muss der Index neu erstellt werden[1]. Stattdessen bietet es sich an einen dynamischen Index einzusetzen, der Aktualisierungen des Bestands ermöglicht – ohne die Notwendigkeit den Index komplett neu zu erstellen (vgl. Büttcher, Clarke und Cormack 2010, Kapitel 7).

6.3 Evaluierung weiterer Ähnlichkeitsmaße

Der Entwurf beschreibt den Ähnlichkeitsvergleich mit Hilfe der Kosinus-Ähnlichkeit. Auch wenn diese gute Ergebnisse liefert, so wäre es interessant alternative Ähnlichkeitsmaße zu evaluieren. Viele dieser Ähnlichkeitsmaße lassen sich ebenfalls durch die Verwendung eines inversen Index optimieren.

[1] Da sich die Idf-Werte für alle Terme ändern, ändert sich auch die Länge aller Dokumentvektoren, welche im Forwärts-Index gespeichert wird.

Interessant erscheint auch die Verwendung von Hybrid-Methoden, also die parallele Verwendung mehrerer Ähnlichkeitsmaße, um möglicherweise die Vorteile verschiedener Methoden zu kombinieren. Dabei ist zu klären, ob sich die Kombination mehrerer Methoden negativ auf die Gesamtleistung auswirkt, oder ob möglicherweise sogar eine Steigerung der Effizienz möglich ist, wenn Kandidaten vorzeitig ausgeschlossen werden können.

Literatur

Balû (Okt. 2005a). *Liste der häufigsten Nachnamen Deutschlands*. URL: http://de.wiktionary.org/w/index.php?title=Wiktionary:Deutsch/Liste_der_h%C3%A4ufigsten_Nachnamen_Deutschlands&direction=prev&oldid=896420 (besucht am 08. 04. 2011).

— (Okt. 2005b). *Liste der häufigsten weiblichen Vornamen Deutschlands*. URL: http://de.wiktionary.org/w/index.php?title=Wiktionary:Deutsch/Liste_der_h%C3%A4ufigsten_weiblichen_Vornamen_Deutschlands&oldid=129897 (besucht am 08. 04. 2011).

— (Sep. 2009). *Liste der häufigsten weiblichen Vornamen Deutschlands*. URL: http://de.wiktionary.org/w/index.php?title=Wiktionary:Deutsch/Liste_der_h%C3%A4ufigsten_m%C3%A4nnlichen_Vornamen_Deutschlands&oldid=1067735 (besucht am 08. 04. 2011).

Bayardo, Roberto J., Yiming Ma und Ramakrishnan Srikant (2007). „Scaling up all pairs similarity search“. In: *Proceedings of the 16th international conference on World Wide Web*. WWW '07. New York, NY, USA: ACM, S. 131–140. ISBN: 978-1-59593-654-7. URL: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.102.2601&rep=rep1&type=pdf.

Büttcher, Stefan, Charles L. A. Clarke und Gordon V. Cormack (2010). *Information Retrieval: Implementing and Evaluating Search Engines*. MIT Press. ISBN: 978-0-262-02651-2. URL: http://ir.uwaterloo.ca/book.

Carpenter, Bob (Dez. 2006). *Code Spelunking: Jaro-Winkler String Comparison*. URL: http://lingpipe-blog.com/2006/12/13/code-spelunking-jaro-winkler-string-comparison/ (besucht am 05. 01. 2011).

Cohen, William W., Pradeep Ravikumar und Stephen E. Fienberg (Aug. 2003). „A Comparison of String Distance Metrics for Name-Matching Tasks“. In: *Proceedings of IJCAI-03 Workshop on Information Integration*, S. 73–78.

Damerau, Fred J. (März 1964). „A technique for computer detection and correction of spelling errors“. In: *Commun. ACM* 7 (3), S. 171–176. ISSN: 0001-0782. URL: http://dedupe.tickett.net/images/c/c0/Acm_march1964.pdf.

De Coninck, Wim (Sep. 2009). *Improved Duplicate Detection in Dynamics CRM*. URL: http://www.orbitone.com/en/blog/archive/2009/09/08/improved-duplicate-detection-in-dynamics-crm.aspx (besucht am 13. 02. 2012).

Draisbach, Uwe und Felix Naumann (Aug. 2009). „A Comparison and Generalization of Blocking and Windowing Algorithms for Duplicate Detection“. In: *International Workshop on Quality in Databases (QDB)*. URL: http://www.hpi.uni-potsdam.de/fileadmin/hpi/FG_Naumann/publications/2009/QDB09_crc.pdf.

Firstlogic Solutions. *Firstlogic Solutions - Data Quality. Delivered.* URL: http://www.firstlogicsolutions.com/ (besucht am 14. 02. 2012).

Gravano, Luis u. a. (2003). „Text joins in an RDBMS for web data integration“. In: *Proceedings of the 12th international conference on World Wide Web*. WWW '03. Budapest, Hungary: ACM, S. 90–101. ISBN: 1-58113-680-3. URL: http://pages.stern.nyu.edu/~panos/publications/www2003.pdf.

Hainke, Thomas. *BatchDeduplicator 4.04 - Die Software mit der Datenqualität planbar wird*. URL: http://www.dataqualityapps.de/dublettensuche.html (besucht am 13. 02. 2012).

Hernández, Mauricio A. und Salvatore J. Stolfo (1995). „The merge/-purge problem for large databases“. In: *Proceedings of the 1995 ACM SIGMOD international conference on Management of data*. SIGMOD '95. New York, NY, USA: ACM, S. 127–138. ISBN: 0-89791-731-6. URL: http://www.cs.washington.edu/education/courses/cse590q/04au/papers/Hernandez95.pdf.

— (Jan. 1998). „Real-world Data is Dirty: Data Cleansing and The Merge/Purge Problem“. In: *Data Min. Knowl. Discov.* 2 (1), S. 9–37. ISSN: 1384-5810. URL: http://blondie.cs.byu.edu/CS652/hernandez98realworld.pdf.

Jaro, Matthew A. (1989). „Advances in Record-Linkage Methodology as Applied to Matching the 1985 Census of Tampa, Florida“. In: *Journal of the American Statistical Association* 84.406, S. 414–420. URL: http://www.fcsm.gov/working-papers/mjaro.pdf.

Kroll-Software. *FuzzyDupes 2011 - Unscharfe Dublettensuche in Datenbanken.* URL: http://www.kroll-software.de/products/fuzzydupes_g.asp (besucht am 13. 02. 2012).

Leser, Ulf und Felix Naumann (2007). *Informationsintegration - Architekturen und Methoden zur Integration verteilter und heterogener Datenquellen.* dpunkt.verlag, S. I–XIII, 1–464.

Levenshtein, Vladimir (1966). „Binary Codes Capable of Correcting Deletions, Insertions and Reversals". In: *Soviet Physics Doklady.* Bd. 10.

Salton, G., A. Wong und C. S. Yang (Nov. 1975). „A vector space model for automatic indexing". In: *Commun. ACM* 18 (11), S. 613–620. ISSN: 0001-0782. URL: http://www.cs.uiuc.edu/class/fa05/cs511/Spring05/other_papers/p613-salton.pdf.

Sarawagi, Sunita und Alok Kirpal (2004). „Efficient set joins on similarity predicates". In: *Proceedings of the 2004 ACM SIGMOD international conference on Management of data.* SIGMOD '04. New York, NY, USA: ACM, S. 743–754. ISBN: 1-58113-859-8. URL: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.100.9714&rep=rep1&type=pdf.

Sundaram, Rangarajan K. (Juni 1996). *A First Course in Optimization Theory.* Cambridge University Press. ISBN: 0521497701. URL: http://www.amazon.com/gp/product/0521497701.

Winkler, William E. (1990). „String Comparator Metrics and Enhanced Decision Rules in the Fellegi-Sunter Model of Record Linkage". In: *Proceedings of the Section on Survey Research.* Wachington, DC, S. 354–359. URL: http://www.amstat.org/sections/srms/Proceedings/papers/1990_056.pdf.

Ich versichere, die von mir vorgelegte Arbeit selbständig verfasst zu haben. Alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten oder nicht veröffentlichten Arbeiten anderer entnommen sind, habe ich als entnommen kenntlich gemacht. Sämtliche Quellen und Hilfsmittel, die ich für die Arbeit benutzt habe, sind angegeben. Die Arbeit hat mit gleichem Inhalt bzw. in wesentlichen Teilen noch keiner anderen Prüfungsbehörde vorgelegen.

(Thomas Krause)

Gummersbach, den 26.02.2012